# Fast Optimal Load Balancing Algorithms for 1D Partitioning[*]

Ali Pınar
Department of Computer Science
University of Illinois at Urbana-Champaign
alipinar@cse.uiuc.edu

Cevdet Aykanat
Computer Engineering Department
Bilkent University, Ankara, Turkey
aykanat@cs.bilkent.edu.tr

**Proposed running head :** Fast Optimal Load Balancing for 1D Partitioning

**Corresponding author :** Assoc. Prof. Cevdet Aykanat
Computer Engineering Department
Bilkent University
TR-06533, Ankara, TURKEY
e-mail: aykanat@cs.bilkent.edu.tr
tel : +90 (312) 290-1625
fax : +90 (312) 266-4126

[*]This work is partially supported by The Scientific and Technical Research Council of Turkey under grant EEEAG-199E001

## Abstract

One-dimensional decomposition of nonuniform workload arrays for optimal load balancing is investigated. The problem has been studied in the literature as "chains-on-chains partitioning" problem. Despite extensive research efforts, heuristics are still used in parallel computing community with the "hope" of good decompositions and the "myth" of exact algorithms being hard to implement and not runtime efficient. The main objective of this paper is to show that using exact algorithms instead of heuristics yields significant load balance improvements with negligible increase in preprocessing time. We provide detailed pseudocodes of our algorithms so that our results can be easily reproduced. We start with a review of literature on chains-on-chains partitioning problem. We propose improvements on these algorithms as well as efficient implementation tips. We also introduce novel algorithms, which are asymptotically and runtime efficient. We experimented with datasets from two different applications: Sparse matrix computations and Direct volume rendering. Experiments showed that the proposed algorithms are 100 times faster than a single sparse-matrix vector multiplication for 64-way decompositions on average. Experiments also verify that load balance can be significantly improved by using exact algorithms instead of heuristics. These two findings show that exact algorithms with efficient implementations discussed in this paper can effectively replace heuristics.

**Key Words :** one-dimensional partitioning; optimal load balancing; chains-on-chains partitioning; dynamic programming; iterative refinement; parametric search; parallel sparse matrix vector multiplication; image-space parallel volume rendering.

## 1 Introduction

In this work, we investigate block partitioning of possibly multi-dimensional nonuniform domains over one-dimensional (1D) workload arrays. The communication and synchronization overhead is assumed to be handled implicitly by the selection of proper partitioning and parallel computation schemes at the beginning, so that load balance is the only metric explicitly considered in the decomposition. The load balancing problem in the partitioning can be modeled as the *chains-on-chains partitioning* (CCP) problem with nonnegative task weights and unweighted edges between successive tasks. The objective of the CCP problem is to find a sequence of $K-1$ separator indices to divide a chain of $N$ tasks with associated computational weights into $K$ consecutive parts such that the *bottleneck value*—the load of the maximally loaded part—is minimized.

The first polynomial time algorithm for solving the CCP problem was proposed by Bokhari [4]. Bokhari's $O(N^3K)$-time algorithm is based on finding a minimum path on a layered graph. Nicol and O'Hallaron [30] reduced the complexity to $O(N^2K)$ by decreasing the number of edges in the layered graph. The algorithm paradigms used in the following works can be classified as *dynamic programming* (DP), *iterative refinement*, and *parametric search*. Anily and Federgruen [1] initiated the DP approach with an $O(N^2K)$-time algorithm. Hansen and Lih [13] independently proposed an $O(N^2K)$-time algorithm. Choi and Narahari [6], and Olstad and Manne [34] introduced asymptotically faster $O(NK)$-time, and $O((N-K)K)$-time DP-based algorithms, respectively. Iterative refinement approach starts with a partition and iteratively tries to improve the solution. The $O((N-K)K\log K)$-time algorithm proposed by Manne and Sørevik [26] falls into this class.

The parametric-search approach relies on repeated probing for the existence of a partition with a *bottleneck* value no greater than a given value. Such a probing takes $\theta(N)$ time since every task has to be examined. Since probing needs to be performed repeatedly, an individual probe can efficiently be performed in $O(K\lg N)$-time through binary search, after performing an initial prefix-sum operation in $\theta(N)$-time for task chains with zero communication costs [17]. Later, $O(K\lg N)$-time probe algorithms were proposed to handle task chains with nonzero communication costs [19, 20, 21, 32]. Finally, the complexity of an individual probe call was reduced to $O(K\lg(N/K))$ by Han, Narahari, and Choi [12].

The parametric-search approach goes back to Iqbal's [16, 20] work describing an $\epsilon$-approximate algorithm, which performs $O(\lg(W_{tot}/\epsilon))$ probe calls. Here, $W_{tot}$ denotes the total task weight and $\epsilon$ denotes the desired accuracy. Iqbal's algorithm exploits the observation that the bottleneck value is in the range $[W_{tot}/K, W_{tot}]$, and performs a binary search in this range by making $O(\lg(W_{tot}/\epsilon))$ probe calls. This work was followed by several exact algorithms involving efficient schemes for the search over bottleneck values by considering only subchain weights. Nicol and O'Hallaron [30, 32] proposed a search scheme which requires at most $4N$ probe calls. Iqbal and Bokhari [21] relaxed the restriction of this algorithm [30, 32] on bounded task weight and communication cost, by proposing a condensation algorithm. Iqbal [18] and Nicol [32, 33] concurrently proposed an efficient search scheme which finds an optimal partition after only $O(K\lg N)$ probe calls.

Asymptotically best algorithms were proposed by Frederickson [7, 8] and Han, Narahari, and Choi [12].

Frederickson proposed an $O(N)$-time optimal algorithm using parametric search. Han et. al. proposed a recursive algorithm with complexity $O(N+K^{1+\varepsilon})$ for any small $\varepsilon > 0$. However, these two works have mostly centered around decreasing the asymptotic running time, disregarding the usefulness of the presented methods in application.

Despite these extensive research efforts on the solution of the CCP problem, heuristics are still commonly used in the parallel computing community. A recent research work [27] exists that is devoted to proposing efficient heuristics for the CCP problem. This attitude depends on the ease of implementation, efficiency, and expectation of "good" quality decompositions of heuristics, and the misconception that exact CCP algorithms are not affordable as a preprocessing step for efficient parallelization. This work proposes efficient exact CCP algorithms. The implementation details and the pseudocodes of the proposed algorithms are clearly presented so that they can easily be reproduced. In order to justify the use of the proposed algorithms, we also demonstrate that qualities of the decompositions obtained through heuristics substantially deviate from those of the optimal ones through experimental results on a wide range of real-world problems.

We run an effective heuristic, as a pre-processing step, to find a "good" upper bound on the optimal bottleneck value. Then, we exploit the lower and upper bounds on the optimal bottleneck value to restrict the search space for separator-index values. This separator-index bounding scheme is exploited in a static manner in the DP algorithm drastically reducing the number of table entries computed and referenced. A dynamic separator-index bounding scheme is proposed for parametric search algorithms. This scheme narrows separator-index ranges after each probe call. The upper bound on the optimal bottleneck value is also exploited to find a much better initial partition for the iterative-refinement algorithm proposed by Manne and Sørevik [26]. We also propose a different iterative-refinement scheme, which is very fast for small to medium number of processors. The observations in the proposed iterative-refinement scheme is further exploited for incorporating the subchain-weight concept into Iqbal's [16, 20] approximate bisection algorithm to make it an exact algorithm.

Two distinct application domains are investigated for experimental performance evaluation of the proposed algorithms. These are 1D decomposition of irregularly sparse matrices for parallel matrix-vector multiplication (SpMxV), and decomposition for image-space parallel volume rendering. SpMxV is the most time consuming operation in iterative solvers, which are widely used for the solution of sparse linear system of equations. Volume rendering is widely used for scientific visualization. Integer and real valued 1D workload arrays arising in the former and latter applications are the distinct features of these two applications. Furthermore, SpMxV, a fine-grain application, is exploited to demonstrate the feasibility of using optimal load balancing algorithms even in sparse-matrix decomposition. Experiments with the proposed CCP algorithms on a wide range of sparse test matrices show that 64-way decompositions can be achieved in 100 times less than a single SpMxV computation time, while producing 4 times better load imbalance values than the most effective heuristic, on average. Experimental results on volume rendering dataset show that exact algorithms can produce 3.8 times better 64-way decompositions than the most effective heuristic, while being only 11 percent slower, on average.

Existing load–balancing models for parallel iterative solvers consider only the SpMxV operations. In this work, we also propose a new load balancing model which considers both the SpMxV and linear vector operations. The proposed model enables the use of the CCP algorithms without any additional overhead.

In this work, we also consider the load–balancing problem for heterogeneous systems. We briefly mention about the modifications needed to enable the use of the proposed CCP algorithms for heterogeneous systems. Finally, we prove the NP-Completeness of the chains partitioning (CP) problem for heterogeneous systems, where processor permutation is allowed in subchain to processor assignment.

The organization of the paper is as follows. Section 2 presents CCP problem definition. A survey on the existing CCP algorithms is presented in Section 3. Proposed CCP algorithms are discussed in Section 4. Load-balancing applications used in experimentations are described in Section 5 and performance results are discussed in Section 6. Appendix A briefly presents proposed load-balancing model for iterative solvers, and Appendix B presents our discussion on CCP and CP problems for heterogeneous systems.

## 2 Preliminaries

In the CCP problem, a computational problem decomposed into a chain $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$ of $N$ task/modules with the associated *positive* computational weights $\mathcal{W} = \langle w_1, w_2, \ldots, w_N \rangle$ is to be mapped onto a chain $\mathcal{P} = \langle P_1, P_2, \ldots, P_K \rangle$ of $K$ *homogeneous* processors. A *subchain* of $\mathcal{T}$ is defined as any subset of contiguous tasks, and the subchain consisting of tasks $\langle t_i, t_{i+1}, \ldots, t_j \rangle$ is denoted as $\mathcal{T}_{i,j}$. Computational load $W_{i,j}$ of subchain $\mathcal{T}_{i,j}$ is equal to $W_{i,j} = \sum_{h=i}^{j} w_h$. From the contiguity constraint, a partition $\Pi$ should map contiguous subchains to contiguous processors. Hence, a $K$-way chain-partition $\Pi_N^K$ of a task chain $\mathcal{T}$ with $N$ tasks onto a processor chain $\mathcal{P}$ with $K$ processors is described by a sequence $\Pi_N^K = \langle s_0, s_1, s_2, \ldots, s_K \rangle$ of $K+1$ separator indices, where $s_0 \leq s_1 \leq s_2 \leq \cdots \leq s_K$ with $s_0 = 0$ and $s_K = N$. Here, $s_k$ denotes the index of the last task of the $k$th part mapped to processor $P_k$ so that $P_k$ gets the subchain $\mathcal{T}_{s_{k-1}+1, s_k}$ with load $L_k = W_{s_{k-1}+1, s_k}$ for $k = 1, 2, \ldots, K$. Hence, partition and mapping will be used interchangeably throughout the paper, since a partition $\Pi$ also defines a mapping. Cost $C(\Pi)$ of a partition $\Pi$ is determined by the maximum processor execution time among all processors, i.e., $C(\Pi) = B = \max_{1 \leq k \leq K} \{L_k\}$. This $B$ value of a partition is called its *bottleneck* value, and the processor/part defining it is called the *bottleneck processor/part*. Hence, the CCP problem can be defined as finding a mapping $\Pi_{opt}$ which minimizes the bottleneck value $B_{opt} = C(\Pi_{opt})$.

## 3 Previous Work on the CCP Problem

Each CCP algorithm discussed in this section and Section 4 involves an initial prefix-sum operation on task-weight array $\mathcal{W}$ for the efficiency of subsequent subchain-weight computations. So, in the discussion of each algorithm, $\mathcal{W}$ is used to refer to the prefix-summed $\mathcal{W}$-array, where cost of this initial prefix-sum operation is considered in the complexity analysis. The presentation of all algorithms focus only on finding the bottleneck value $B_{opt}$ of optimal partition(s). An optimal solution can be easily and efficiently constructed by making a PROBE($B_{opt}$) call discussed in Section 3.4 after finding $B_{opt}$. This approach avoids the overhead of maintaining additional information during the course of the algorithm needed to ease the construction of an optimal partition.

## 3.1 Heuristics

Most commonly used heuristic is based on *recursive bisection* (RB). RB achieves $K$-way partitioning through $\lg K$ bisection levels, where $K$ is a power of 2. At each bisection step in a level, the current chain is divided evenly into two subchains. Although optimal division can easily be achieved at every bisection step, the sequence of optimal bisections may lead to poor load balancing. RB can be efficiently implemented in $O(N+K\lg N)$ time, by first performing a prefix-sum operation on the workload array $\mathcal{W}$, with complexity $O(N)$, and then making $K-1$ binary searches in the prefix-summed $\mathcal{W}$-array each with complexity $O(\lg N)$.

Miguet and Pierson [27] recently proposed two other heuristics. The first heuristic (H1) computes the separator values such that $s_k$ is the largest index where $W_{s_k} \leq kB^*$. Here, $B^* = W_{tot}/K$ is the ideal bottleneck value, and $W_{tot} = \sum_{i=1}^{N} w_i$ denotes the sum of all task weights. The second heuristic (H2), further refines the separator indices by incrementing each $s_k$ value found in H1 if $(W_{s_{k+1}} - kB^*) < (kB^* - W_{s_k})$. These two heuristics can also be implemented in $O(N+K\lg N)$ time, by performing $K-1$ binary searches in the prefix-summed $\mathcal{W}$-array.

Miguet and Pierson [27] have already proved the upper bounds on the bottleneck values of the partitions found by H1 and H2 as $B_{H1}, B_{H2} < B^* + w_{max}$, where $w_{max} = \max_{1\leq k\leq N}\{w_i\}$ denotes the maximum task weight. The following lemma establishes a similar bound for the RB heuristic.

LEMMA 1. *Let $\Pi_{RB} = \langle s_0, s_1, \ldots, s_K\rangle$ be a partition constructed by the RB heuristic for a given CCP problem instance $(\mathcal{W}, N, K)$. Then, $B_{RB} = C(\Pi_{RB})$ satisfies $B_{RB} \leq B^* + w_{max}(K-1)/K$.*

PROOF. Consider the first bisection step. There exists an index $1 \leq i_1 \leq N$ such that both $W_{1,i_1-1}, W_{i_1+1,N} \leq W_{tot}/2$ and both $W_{1,i_1-1} + w_{i_1} = W_{1,i_1}$, $W_{i_1+1,N} + w_{i_1} = W_{i_1,N} \geq W_{tot}/2$. The worst case for RB occurs when $w_{i_1} = w_{max}$ and $W_{1,i_1-1} = W_{i_1+1,N} = (W_{tot} - w_{max})/2$. Without loss of generality, assume that $t_{i_1}$ is assigned to the left part so that $s_{K/2} = i_1$ and $W_{1,s_{K/2}} = W_{tot}/2 + w_{max}/2$. In a similar worst-case bisection of $\mathcal{T}_{1,s_{K/2}}$, there exists an index $i_2$ such that $w_{i_2} = w_{max}$ and $W_{1,i_2-1} = W_{i_2+1,s_{K/2}} = (W_{tot} - w_{max})/4$, and $t_{i_2}$ is assigned to the left part so that $s_{K/4} = i_2$ and $W_{1,s_{K/4}} = (W_{tot} - w_{max})/4 + w_{max} = W_{tot}/4 + (3/4)w_{max}$. For a sequence of $\lg K$ such worst case bisection steps on the left parts, processor $P_1$ will be the bottleneck processor with load $B_{RB} = W_{1,s_1} = W_{tot}/K + w_{max}(K-1)/K$. $\square$

## 3.2 Dynamic Programming

The *overlapping* subproblem space can be defined as $\mathcal{T}_i^k$, for $k = 1, 2, \ldots, K$ and $i = k, k+1, \ldots, N-K+k$, where $\mathcal{T}_i^k$ denotes the $k$-way CCP of the prefix task-subchain $\mathcal{T}_{1,i} = \langle t_1, t_2, \ldots, t_i\rangle$ onto the prefix processor-subchain $\mathcal{P}_{1,k} = \langle P_1, P_2, \ldots, P_k\rangle$. The lower and upper bounds on index $i$ for a particular $k$ are because of the fact that there is no merit in leaving a processor empty in any mapping. From this subproblem space definition, the *optimal substructure* property of the CCP problem can be shown by considering an optimal mapping $\Pi_i^k = \langle s_0, s_1, \ldots, s_k = i\rangle$ with a bottleneck value $B_i^k = C(\Pi_i^k)$ for the CCP subproblem $\mathcal{T}_i^k$. If the last processor is not the bottleneck processor in $\Pi_i^k$, then $\Pi_{s_{k-1}}^{k-1} = \langle s_0, s_1, \ldots, s_{k-1}\rangle$ should be an optimal mapping

```
DP ($\mathcal{W}, N, K$)
  $B[1,i] \leftarrow \mathcal{W}[i]$ for $i = 1, 2, \ldots, N$;
  for $k \leftarrow 2$ to $K$ do
    $j \leftarrow k-1$;
    for $i \leftarrow k$ to $N-K+k$ do
      if $\mathcal{W}[i] - \mathcal{W}[j] > B[k-1, j]$ then
        repeat $j \leftarrow j+1$ until $\mathcal{W}[i] - \mathcal{W}[j] \leq B[k-1, j]$;
        if $\mathcal{W}[i] - \mathcal{W}[j-1] < B[k-1, j]$ then
          $j \leftarrow j-1$;
          $B[k, i] \leftarrow \mathcal{W}[i] - \mathcal{W}[j]$;
        else
          $B[k, i] \leftarrow B[k-1, j]$;
      else
        $B[k, i] \leftarrow B[k-1, j]$;
  return $B_{opt} \leftarrow B[K, N]$;
```

Figure 1: $O((N-K)K)$-time dynamic-programming algorithm proposed by Choi and Narahari [6], and Olstad and Manne [34].

for the subproblem $\mathcal{T}^{k-1}_{s_{k-1}}$. Hence, the recursive definition for the bottleneck value of an optimal mapping becomes

$$B^k_i = \min_{k-1 \leq j < i} \left\{ \max\{B^{k-1}_j, W_{j+1,i}\} \right\}. \tag{1}$$

In (1), search for index $j$ corresponds to search for separator $s_{k-1}$ so that remaining subchain $\mathcal{T}_{j+1,i}$ is assigned to the last processor $P_k$ in an optimal mapping $\Pi^k_i$ of $\mathcal{T}^k_i$. The bottleneck value $B^K_N$ of an optimal mapping can be computed using (1) in a bottom-up fashion starting from $B^1_i = W_{1,i}$ for $i = 1, 2, \ldots, N$. An initial prefix-sum on the workload array $\mathcal{W}$ enables the constant-time computation of the subchain weight of the form $W_{j+1,i}$ through $W_{j+1,i} = \mathcal{W}[i] - \mathcal{W}[j]$. Computing $B^k_i$ using (1) takes $O(N-k)$ time for each $i$ and $k$, and thus the algorithm takes $O((N-K)^2 K)$ time since the number of distinct subproblems is equal to $(N-K+1)K$.

Choi and Narahari [6], and Olstad and Manne [34] reduced the complexity of this scheme to $O(NK)$ and $O((N-K)K)$, respectively, by exploiting the following observations that hold for positive task weights. For a fixed $k$ in (1), the minimum index value $j^k_i$ defining $B^k_i$ cannot occur at a value less than the minimum index value $j^k_{i-1}$ defining $B^k_{i-1}$, i.e., $j^k_{i-1} \leq j^k_i \leq (i-1)$. Hence, the search for optimal $j^k_i$ can start from $j^k_{i-1}$. In (1), $B^{k-1}_j$ for a fixed $k$ is a nondecreasing function of $j$, and $W_{j+1,i}$ for a fixed $i$ is a decreasing function of $j$ reducing to 0 at $j = i$. So, two distinct cases occur in semi-closed interval $[j^k_{i-1}, i)$ for $j$. If $W_{j+1,i} > B^{k-1}_j$ initially then these two functions intersect in $[j^k_{i-1}, i)$. In this case, search for $j^k_i$ continues until $W_{j^*+1,i} \leq B^{k-1}_j$ and then only $j^*$ and $j^*-1$ are considered for setting $j^k_i$ as $j^k_i = j^*$ if $B^{k-1}_{j^*} \leq W_{j,i}$ and $j^k_i = j^*-1$ otherwise. Note that this scheme automatically detects $j^k_i = i-1$ if $W_{j+1,i}$ and $B^{k-1}_j$ intersect in open interval $(i-1, i)$. If, however, $W_{j+1,i} \leq B^{k-1}_j$ initially then $B^{k-1}_j$ lies above $W_{j+1,i}$ in closed interval $[j^k_{i-1}, i]$. In this case, the minimum value occurs at the first value of $j$, i.e., $j^k_i = j^k_{i-1}$. These improvements lead to an $O((N-K)K)$-time algorithm since the computation of all $B^k_i$ values for a fixed $k$ makes $O(N-K)$ references to already computed $B^{k-1}_j$ values. Fig. 1 displays a run-time efficient implementation scheme of this $O((N-K)K)$-time DP algorithm, which avoids the explicit min-max operation required in (1). In Fig. 1, $B^k_i$ values are stored in a table whose entries are computed in row–major order.

```
MS ($\mathcal{W}, N, K$)
  $s_k \leftarrow k$ for $k \leftarrow 0, 1, \ldots, K-1$;  $s_K \leftarrow N$;
  $L_k \leftarrow w_k$ for $k \leftarrow 1, \ldots, K-1$;  $L_K \leftarrow \sum_{k=K}^{N} w_k$;
  repeat
    $b \leftarrow \{ k \mid L_k$ is maximum over $1 \le k \le K\}$;
    $B \leftarrow L_b$;
    if $s_{b-1} + 1 = s_b$ then
      exit the repeat-loop;
    $k \leftarrow b$;
    while $L_k \ge B$ and $k > 1$ do
      $s_{k-1} \leftarrow s_{k-1} + 1$;
      $L_k \leftarrow L_k - w_{s_{k-1}}$;
      $L_{k-1} \leftarrow L_{k-1} + w_{s_{k-1}}$;
      if $L_k \le B$ then
        $k \leftarrow k - 1$;
  until $L_1 \ge B$;
  return $B_{opt} \leftarrow B$;
```

Figure 2: Iterative refinement algorithm proposed by Manne and Sorevik [26].

## 3.3 Iterative Refinement

The algorithm proposed by Manne and Sorevik [26], referred to here as the MS algorithm, is based on finding a sequence of non-optimal partitions such that there exists only one way each partition can be improved. For this purpose, they introduce a special kind of partition, namely the *leftist partition (LP)*. Consider a partition $\Pi$ such that $P_k$ is the leftmost processor containing at least two tasks. $\Pi$ is defined as an LP if increasing the load of any processor $P_\ell$ that lies to the right of $P_k$ by augmenting the last task of $P_{\ell-1}$ to $P_\ell$ makes $P_\ell$ a bottleneck processor with a load greater than or equal to $C(\Pi)$.

Let $\Pi$ be an LP with bottleneck processor $P_b$ and bottleneck value $C(\Pi) = B = L_b$. If $P_b$ contains only one task then $\Pi$ is optimal. So, assume that $P_b$ contains at least two tasks. The refinement step, which is shown by the inner while-loop in Fig. 2, then tries to find a new LP of lower cost by successively removing the first task of $P_k$ and augmenting it to $P_{k-1}$ for $k = b, b-1, \ldots$ until $L_k < B$. Unsuccessful refinement occurs when the while-loop proceeds until $k = 1$ with $L_k \ge B$. Manne and Sorevik [26] proved that a successful refinement of an LP gives a new LP and that LP must be optimal if the refinement is unsuccessful. So, an initial LP is needed to start the algorithm. As shown in Fig. 2, Manne and Sorevik [26] proposed to use an initial LP where the $K-1$ leftmost processors each have only one task and last processor contains the rest of the tasks.

The MS algorithm moves each separator index at most $N - K$ times so that the total number of separator-index moves is $O(K(N-K))$. A max-heap is maintained for the processor loads to speed up the operation of finding a bottleneck processor at the beginning of each repeat-loop iteration. The cost of each separator-index move is $O(\lg K)$ since it necessitates one decrease-key and one increase-key operations. So, the overall complexity of the MS algorithm is $O(K(N-K)\lg K)$.

## 3.4 Parametric Search

The parametric-search approach relies on repeated probing for the existence of a partition $\Pi$ with a bottleneck value no greater than a given $B$ value, i.e., $C(\Pi) \le B$. The probe algorithms exploit the *greedy-choice* property on the existence and construction of $\Pi$. The greedy choice here is to minimize the work remaining after loading processor $P_k$ subject to $L_k \le B$ for $k = 1, \ldots, K-1$ in order. The PROBE($B$) functions given in Fig. 3 exploit

```
PROBE $(B)$
  $s_0 \leftarrow 0$;   $k \leftarrow 1$;
  $Bsum \leftarrow B$;
  while $k \leq K$ and $Bsum < W_{tot}$ do
    $s_k \leftarrow$ BINSRCH $(\mathcal{W}, s_{k-1}+1, N, Bsum)$;
    $Bsum \leftarrow \mathcal{W}[s_k] + B$;
    $k \leftarrow k+1$;
  if $Bsum \geq W_{tot}$ then
    return TRUE;
  else
    return FALSE;
```

(a)

```
PROBE $(B)$
  $s_0 \leftarrow 0$;   $k \leftarrow 1$;   $step \leftarrow N/K$;
  $Bsum \leftarrow B$;
  while $k \leq K$ and $Bsum < W_{tot}$ do
    while $\mathcal{W}[step] < Bsum$ do
      $step \leftarrow step + N/K$;
    $s_k \leftarrow$ BINSRCH $(\mathcal{W}, step - N/K, step, Bsum)$;
    $Bsum \leftarrow \mathcal{W}[s_k] + B$;
    $k \leftarrow k+1$;
  if $Bsum \geq W_{tot}$ then
    return TRUE;
  else
    return FALSE;
```

(b)

Figure 3: (a) Standard probe algorithm with $O(K \lg_2 N)$ complexity, (b) $O(K \lg_2(N/K))$-time probe algorithm proposed by Han, Narahari, and Choi [12].

this greedy property as follows. PROBE finds the largest index $s_1$ so that $W_{1,s_1} \leq B$, and assigns subchain $\mathcal{T}_{1,s_1}$ to processor $P_1$ with load $L_1 = W_{1,s_1}$. Hence, the first task in the second processor is $t_{s_1+1}$. PROBE then similarly finds the largest index $s_2$ so that $W_{s_1+1,s_2} \leq B$, and assigns the subchain $\mathcal{T}_{s_1+1,s_2}$ to processor $P_2$ with load $L_1 = W_{s_1+1,s_2}$. This process continues until either all tasks are assigned or the processors are exhausted. The former and latter cases denote the feasibility and infeasibility of $B$, respectively.

Fig. 3(a) illustrates the standard probe algorithm. As seen in Fig. 3, the indices $s_1, s_2, \ldots s_{K-1}$ are efficiently found through binary search (BINSRCH) on the prefix-summed $\mathcal{W}$-array. In this figure, BINSRCH$(\mathcal{W}, i, N, Bsum)$ searches $\mathcal{W}$ in the index range $[i, N]$ to compute the index $i \leq j \leq N$ such that $\mathcal{W}[j] \leq Bsum$ and $\mathcal{W}[j+1] > Bsum$. The complexity of the standard probe algorithm is $\sum_{k=0}^{K-1} \theta(\lg(N - s_k)) = O(K \lg N)$. Han, Narahari and Choi [12] proposed an $O(K \lg N/K)$-time probe algorithm (see Fig. 3(b)) exploiting $K$ repeated binary searches on the same $\mathcal{W}$-array with increasing search values. Their algorithm divides the chain into $K$ subchains of equal length. At each probe call, a linear search is performed on the weights of the last tasks of these $K$ subchains to find out in which subchain the search value could be, and then binary search is performed on the respective subchain of length $N/K$. Note that since the probe search values always increase, linear search can be performed incrementally, that is search continues from the last subchain that was searched to the right. Thus, the total time for $K$ searches on $K$ numbers is only $O(K)$. This means a total cost of $O(K \lg(N/K) + K)$ for $K$ binary searches, and hence the probe function.

### 3.4.1 Bisection as an Approximation Algorithm

Let $f(B)$ be the binary-valued function where $f(B) = 1$ if PROBE(B) is true and $f(B) = 0$ if PROBE(B) is false. It is clear that $f(B)$ is nondecreasing in $B$. It is also clear that $B_{opt}$ lies between $LB = B^* = W_{tot}/K$ and $UB = W_{tot}$. These observations are exploited in the *bisection* algorithm leading to an efficient $\epsilon$-approximate algorithm, where $\epsilon$ is the desired precision. Interval $[W_{tot}/K, W_{tot}]$ is conceptually discretized into $(W_{tot} - W_{tot}/K)/\epsilon$ bottleneck values, and binary search is used in this range to find the minimum feasible bottleneck value $B_{opt}$. Fig. 4 illustrates the bisection algorithm. The bisection algorithm performs $O(\lg(W_{tot}/\epsilon))$ PROBE calls and each PROBE call costs $O(K \lg(N/K))$. Hence, the bisection algorithm runs in $O(N + K \lg(N/K) \lg(W_{tot}/\epsilon))$ time, where $\theta(N)$ cost comes from the initial prefix-sum operation on $\mathcal{W}$. The performance of this algorithm deteriorates when $\lg(W_{tot}/\epsilon)$ becomes comparable with $N$.

```
ε-BISECT ($\mathcal{W}, N, K, \epsilon$)
  $LB \leftarrow B^*$;  $UB \leftarrow W_{tot}$;
  repeat
    $B \leftarrow (UB + LB) / 2$;
    if PROBE ($B$) then
      $UB \leftarrow B$;
    else
      $LB \leftarrow B$;
  until $UB \leq LB + \epsilon$;
  return $B_{opt} \leftarrow UB$;
```

Figure 4: Bisection as an $\epsilon$-approximation algorithm.

### 3.4.2 Nicol's Algorithm

Nicol's algorithm [33] exploits the fact that any candidate $B$ value is equal to weight $W_{i,j}$ of a subchain. The naive solution is to generate all subchain weights of the form $W_{i,j}$, sort them, and then use binary search to find the minimum $W_{a,b}$ value for which PROBE($W_{a,b}$) = TRUE. Nicol's algorithm efficiently searches for the earliest range $W_{a,b}$ for which $B_{opt} = W_{a,b}$ by considering each processor in order as a candidate bottleneck processor in an optimal mapping. Let $\Pi_{opt}$ be the optimal mapping constructed by greedy PROBE($B_{opt}$), and let processor $P_b$ be the first bottleneck processor with load $L_b = W_{s_{b-1}+1,s_b} = B_{opt}$ in $\Pi_{opt} = \langle s_0, s_1, \ldots, s_{b-1}, s_b, \ldots, s_K \rangle$. Under these assumptions, this greedy construction of $\Pi_{opt}$ ensures that each processor $P_k$ preceding $P_b$ is loaded as much as possible with $L_k < B_{opt}$, for $k = 1, 2, \ldots, b-1$ in $\Pi_{opt}$. Here, PROBE($L_k$) = FALSE since $L_k < B_{opt}$, and PROBE($L_k + w_{s_k+1}$) = TRUE since adding one more task to processor $P_k$ increases its load to $L_k + w_{s_k+1} > B_{opt}$. Hence, if $b = 1$ (i.e., $P_1$ is a bottleneck processor) then $s_1$ is equal to the smallest index $i_1$ such that PROBE($W_{1,i_1}$) = TRUE, and $B_{opt} = B_1 = W_{1,s_1}$. If, however, $b > 1$ then because of the greedy choice property $P_1$ should be loaded as much as possible without exceeding $B_{opt} = B_b < B_1$, which implies that $s_1 = i_1 - 1$ and hence $L_1 = W_{1,i_1-1}$. If $b = 2$ then $s_2$ is equal to the smallest index $i_2$ such that PROBE($W_{i_1,i_2}$) = TRUE, and $B_{opt} = B_2 = W_{i_1,i_2}$. If $b > 2$ then $s_2 = i_2 - 1$. This iterative process continues to compute $i_b$ as the smallest index for which PROBE($W_{i_{b-1},i_b}$) = TRUE and save $B_b = W_{i_{b-1},i_b}$, for $b = 1, 2, \ldots, K-1$ with $i_K = N$. Finally, optimal bottleneck value is selected as $B_{opt} = \min_{1 \leq b \leq K} B_b$.

Fig. 5 illustrates Nicol's algorithm. As seen in this figure, given $i_{b-1}$, $i_b$ is found by performing a binary search over all subchain weights of the form $W_{i_{b-1},j}$, for $i_{b-1} \leq j \leq N$, in the $b$th iteration of the *for-loop*. Hence, Nicol's algorithm performs $O(\lg N)$ PROBE calls to find $i_b$ at iteration $b$, and each PROBE call costs $O(K \lg(N/K))$. Thus, the cost of computing an individual $B_b$ value is $O(K \lg N \lg(N/K))$. Since $K-1$ such $B_b$ values are computed, the overall complexity of Nicol's algorithm is $O(N + K^2 \lg N \lg(N/K))$, where the $\theta(N)$ cost comes from the initial prefix-sum operation on $\mathcal{W}$.

Two possible implementations of Nicol's algorithm are presented in Fig. 5. Fig. 5(a) illustrates a straightforward implementation. Fig. 5(b) illustrates a careful implementation, which maintains and exploits the information on the success or failure of previous probe calls to be able to determine the results of some future probes without invoking the PROBE function. As seen in Fig. 5(b), this information is efficiently maintained as an undetermined bottleneck-value range $(LB, UB)$, which is dynamically refined in the *while-loop* depending on the success or failure of the probe call. Any bottleneck value encountered outside the current range is immediately accepted or rejected without any probe calls. Although this simple scheme does not improve the complexity of the algorithm, it drastically reduces the number of probe calls thus drastically increasing the run-time performance as will be discussed in Section 6.

```
NICOL- $(\mathcal{W}, N, K)$
  $i_0 \leftarrow 1$;
  for $b \leftarrow 1$ to $K-1$ do
    $ilow \leftarrow i_{b-1}$;  $ihigh \leftarrow N$;
    while $ilow < ihigh$ do
      $imid \leftarrow (ilow + ihigh)\ /\ 2$;
      $B \leftarrow \mathcal{W}[imid] - \mathcal{W}[i_{b-1}-1]$;
      if PROBE $(B)$ then
        $ihigh \leftarrow imid$;
      else
        $ilow \leftarrow imid + 1$;
    $i_b \leftarrow ihigh$;
    $B_b \leftarrow \mathcal{W}[i_b] - \mathcal{W}[i_{b-1}-1]$;
  $B_K \leftarrow \mathcal{W}[N] - \mathcal{W}[i_{K-1}-1]$;
  return $B_{opt} \leftarrow \min_{1 \le b \le K}\{B_k\}$;
```

(a)

```
NICOL $(\mathcal{W}, N, K)$
  $i_0 \leftarrow 1$;  $LB \leftarrow B^*$;  $UB \leftarrow W_{tot}$;
  for $b \leftarrow 1$ to $K-1$ do
    $ilow \leftarrow i_{b-1}$;  $ihigh \leftarrow N$;
    while $ilow < ihigh$ do
      $imid \leftarrow (ilow + ihigh)\ /\ 2$;
      $B \leftarrow \mathcal{W}[imid] - \mathcal{W}[i_{b-1}-1]$
      if $LB \le B < UB$ then
        if PROBE $(B)$ then
          $ihigh \leftarrow imid$;
          $UB \leftarrow B$;
        else
          $ilow \leftarrow imid + 1$;
          $LB \leftarrow B$;
      elseif $B \ge UB$
        $ihigh \leftarrow imid$;
      else
        $ilow \leftarrow imid + 1$;
    $i_b \leftarrow ihigh$;
    $B_b \leftarrow \mathcal{W}[i_b] - \mathcal{W}[i_{b-1}-1]$;
  $B_K \leftarrow \mathcal{W}[N] - \mathcal{W}[i_{K-1}-1]$;
  return $B_{opt} \leftarrow \min_{1 \le b \le K}\{B_k\}$;
```

(b)

Figure 5: Nicol's [33] algorithm: (a) straightforward implementation, (b) careful implementation with dynamic bottleneck-value bounding.

## 4 Proposed CCP Algorithms

### 4.1 Restricting the Search Space

The proposed CCP algorithms exploit lower and upper bounds on the optimal bottleneck value to restrict the search space for $s_k$ separator values as a preprocessing step. Natural lower and upper bounds for the optimal bottleneck value $B_{opt}$ of a given CCP problem instance $(\mathcal{W}, N, K)$ are $LB = \max\{B^*, w_{max}\}$ and $UB = B^* + w_{max}$, respectively. However, $w_{max} < B^*$ in coarse grain parallelization ($K \ll N$) of most real-world applications. The presentation, here and hereafter, will be for $w_{max} < B^* = LB$ even though all findings to be presented also become valid by replacing $B^*$ with $LB = \max\{B^*, w_{max}\}$. The following lemma describes how to use these natural bounds on $B_{opt}$ to restrict the search space for the separator values.

LEMMA 2. *For a given CCP problem instance $(\mathcal{W}, N, K)$, if $B_f$ is a feasible bottleneck value in the range $[B^*, B^*+w_{max}]$, then there exists a partition $\Pi = \langle s_0, s_1, \ldots, s_K \rangle$ of cost $C(\Pi) \le B_f$ with $SL_k \le s_k \le SH_k$, for $k = 1, 2, \ldots, K-1$, where $SL_k$ and $SH_k$ are, respectively, the smallest and largest indices such that*

$$W_{1,SL_k} \;\ge\; k\ (B^* - w_{max}(K-k)/K) \quad \text{and} \quad W_{1,SH_k} \;\le\; k\ (B^* + w_{max}(K-k)/K).$$

PROOF. Let $B_f = B^* + w$, where $0 \le w < w_{max}$. Partition $\Pi$ can be constructed by PROBE(B), which loads the first $k$ processors as much as possible subject to $L_q \le B_f$, for $q = 1, 2, \ldots, k$. In the worst case, $w_{s_k+1} = w_{max}$ for each of the first $k$ processors. So, we have $W_{1,s_k} \ge f(w) = k(B^* + w - w_{max})$ for $k = 1, 2, \ldots, K-1$. However, it should be possible to divide the remaining subchain $\mathcal{T}_{s_k+1,N}$ into $K-k$ parts without exceeding $B_f$, i.e., $W_{s_{k+1},N} \le (K-k)\,(B^*+w)$. So, we also have $W_{1,s_k} \ge g(w) = W_{tot} - (K-k)\,(B^*+w)$. Note that $f(w)$ is an increasing, whereas $g(w)$ is a decreasing function of $w$. Hence, the minimum of $\max\{f(w), g(w)\}$ occurs at the intersection of $f(w)$ and $g(w)$, thus $W_{1,s_k} \ge k\ (B^* - w_{max}(K-k)/K)$.

For the proof of the upper bounds on $s_k$ values, we can start with $W_{1,s_k} \le f(w) = k(B^* + w)$ which holds when $L_q = B^* + w$ for $q = 1, 2, \ldots k$. However, the condition $W_{s_k+1,N} \geq (K-k)(B^* + w - w_{max})$ ensures the feasibility of $B_f = B^* + w$, since PROBE(B) can always load each of the remaining $(K-k)$ processors with $B^* + w - w_{max}$. So, we also have $W_{1,s_k} \leq g(w) = W_{tot} - (K-k)(B^* + w - w_{max})$. Here, $f(w)$ is an increasing, whereas $g(w)$ is a decreasing function of $w$. By following similar steps, we can obtain $W_{1,s_k} \leq k\,(B^* + w_{max}(K-k)/K)$. □

COROLLARY 3. *The separator range weights are* $\Delta W_k = \sum_{i=SL_k}^{SH_k} w_i = W_{1,SH_k} - W_{1,SL_k} = 2\,w_{max}\,k\,(K-k)/K$ *with a maximum value* $K w_{max}/2$ *at* $k = K/2$.

The use of this corollary requires finding $w_{max}$, which brings an overhead equivalent to that of the prefix-sum operation, hence should be avoided. In this work, we propose and adopt a practical scheme to construct the bounds on separator indices. We run the RB heuristic to find a hopefully good bottleneck value $B_{RB}$, and use $B_{RB}$ as an upper bound for bottleneck values, i.e., $UB = B_{RB}$. Then, we run LR-PROBE($B_{RB}$) and RL-PROBE($B_{RB}$) to construct two mappings $\Pi^1 = \langle h_0^1, h_1^1, \ldots, h_K^1 \rangle$ and $\Pi^2 = \langle \ell_0^2, \ell_1^2, \ldots, \ell_K^2 \rangle$ with $C(\Pi^1), C(\Pi^2) \leq B_{RB}$. Here, LR-PROBE denotes the left-to-right probe given in Fig. 3, whereas RL-PROBE denotes a right-to-left probe function which can be considered as the dual of the LR-PROBE. RL-PROBE exploits the greedy-choice property from right to left. That is, RL-PROBE assigns subchains from the right end towards the left end of the task chain to processors in the order $P_K, P_{K-1}, \ldots, P_1$. From these two mappings, lower and upper bound values for $s_k$ separator indices are constructed as $SL_k = \ell_k^2$ and $SH_k = h_k^1$, respectively, for $k = 1, \ldots, K-1$. These bounds are further refined by running LR-PROBE($B^*$) and RL-PROBE($B^*$) to construct two mappings $\Pi^3 = \langle \ell_0^3, \ell_1^3, \ldots, \ell_K^3 \rangle$ and $\Pi^4 = \langle h_0^4, h_1^4, \ldots, h_K^4 \rangle$, and then defining $SL_k = \max\{SL_k, \ell_k^3\}$ and $SH_k = \min\{SH_k, h_k^4\}$ for $k = 1, \ldots, K-1$. Lemmas 4 and 5 prove the correctness of these bounds.

LEMMA 4. *For a given CCP problem instance* $(\mathcal{W}, N, K)$ *and a feasible bottleneck value* $B_f$, *let* $\Pi^1 = \langle h_0^1, h_1^1, \ldots, h_K^1 \rangle$, $\Pi^2 = \langle \ell_0^2, \ell_1^2, \ldots, \ell_K^2 \rangle$ *be the partitions constructed by* LR-PROBE($B_f$), *and* RL-PROBE($B_f$), *respectively. Then, any partition* $\Pi = \langle s_0, s_1, \ldots, s_K \rangle$ *of cost* $C(\Pi) = B \leq B_f$ *satisfies* $\ell_k^2 \leq s_k \leq h_k^1$.

PROOF. By the property of LR-PROBE($B_f$) $h_k^1$ is the largest index where $\mathcal{T}_{1,h_k^1}$ can be partitioned into $k$ parts without exceeding $B_f$. If $s_k > h_k^1$, then the bottleneck value will exceed $B_f$ and thus $B$. By the property of RL-PROBE($B_f$) $\ell_k^2$ is the smallest index where $\mathcal{T}_{\ell_k^2,N}$ can be partitioned into $K-k$ parts without exceeding $B_f$. If $s_k < \ell_k^2$, then the bottleneck value will exceed $B_f$ and thus $B$. □

LEMMA 5. *For a given CCP problem instance* $(\mathcal{W}, N, K)$, *let* $\Pi^3 = \langle \ell_0^3, \ell_1^3, \ldots, \ell_K^3 \rangle$, $\Pi^4 = \langle h_0^4, h_1^4, \ldots, h_K^4 \rangle$ *be the partitions constructed by* LR-PROBE($B^*$), *and* RL-PROBE($B^*$), *respectively. Then, for any feasible bottleneck value* $B_f$, *there exists a partition* $\Pi = \langle s_0, s_1, \ldots, s_K \rangle$ *of cost* $C(\Pi) \leq B_f$ *which satisfies* $\ell_k^3 \leq s_k \leq h_k^4$.

PROOF. Consider the partition $\Pi = \langle s_0, s_1, \ldots, s_K \rangle$ constructed by LR-PROBE($B_f$). It is clear that this partition already satisfies the lower bounds, i.e., $s_k \geq \ell_k^3$. Assume that $s_k > h_k^4$, then the partition $\Pi'$ obtained by moving $s_k$ back to $h_k^4$ also yields a partition with cost $C(\Pi') \leq B_f$, since $\mathcal{T}_{h_k^4+1,N}$ can be partitioned into $K-k$ parts without exceeding $B^*$. □

The difference between Lemmas 4 and 5 is that the former ensures the existence of all partitions with cost less than or equal to $B_f$ within the given separator-index ranges, whereas the latter only ensures the existence of at least one such partition within the given ranges. The following corollary combines the results of these two lemmas.

COROLLARY 6. *For a given CCP problem instance $(\mathcal{W}, N, K)$ and a feasible bottleneck value $B_f$, let $\Pi^1 = \langle h_0^1, h_1^1, \ldots, h_K^1\rangle, \Pi^2 = \langle \ell_0^2, \ell_1^2, \ldots, \ell_K^2\rangle, \Pi^3 = \langle \ell_0^3, \ell_1^3, \ldots, \ell_K^3\rangle$, and $\Pi^4 = \langle h_0^4, h_1^4, \ldots, h_K^4\rangle$ be the partitions constructed by* LR-PROBE$(B_f)$, RL-PROBE$(B_f)$, LR-PROBE$(B^*)$, *and* RL-PROBE$(B^*)$, *respectively. Then, for any feasible bottleneck value $B$ in the range $[B^*, B_f]$, there exists a partition $\Pi = \langle s_0, s_1, \ldots, s_K\rangle$ of cost $C(\Pi) \le B$ with $SL_k \le s_k \le SH_k$, for $k = 1, 2, \ldots, K{-}1$, where $SL_k = \max\{\ell_k^2, \ell_k^3\}$ and $SH_k = \min\{h_k^1, h_k^4\}$.*

COROLLARY 7. *The separator range weights become $\Delta W_k = 2\,\min\{k, (K-k)\} w_{max}$ in the worst case, with a maximum value $K w_{max}$ at $k = K/2$.*

Lemma 2 and Corollary 6 directly infer the following theorem since $B^* \le B_{opt} \le B^* + w_{max}$.

THEOREM 8. *For a given CCP problem instance $(\mathcal{W}, N, K)$, and $SL_k$ and $SH_k$ index bounds constructed according to Lemma 2 or Corollary 6, there exists an optimal partition $\Pi_{opt} = \langle s_0, s_1, \ldots, s_K\rangle$ with $SL_k \le s_k \le SH_k$, for $k = 1, 2, \ldots, K-1$.*

Comparison of the separator range weights given in Lemma 2 and Corollary 6 shows that the separator range weights produced by the practical scheme described in Corollary 6 may be twice worse than those of Lemma 2 in the worst-case. However, this is only the worst-case behavior, and the practical scheme finds fairly better bounds since the ordering of the chain usually prevents the worst-case behavior, and $B_{RB} < B^* + w_{max}$. Experimental results given in Section 6 justify this expectation.

### 4.1.1 Complexity Analysis Models

Corollaries 3 and 7 give bounds on the weights of the separator-index ranges. However, we need bounds on the sizes of these separator-index ranges for the sake of computational complexity analysis of the proposed CCP algorithms. Here, the size $\Delta S_k = SH_k - SL_k + 1$ denotes the number of tasks within the $k$th range $[SL_k, SH_k]$.

Miguet and Pierson [27] propose the model $w_i = \theta(w_{avg})$ for $i = 1, 2, \ldots N$ in order to prove that their H1 and H2 heuristics allocate $\theta(N/K)$ tasks to each processor. Here, $w_{avg} = W_{tot}/N$ denotes the average task weight. This assumption means that the weight of each task is not too far away from the average task weight. Using Corollaries 3 and 7, $w_i = \Omega(w_{avg})$ part of the model induces $\Delta S_k = O(K\, w_{max}/w_{avg})$. Moreover, $w_i = O(w_{avg})$ part of the model can be exploited to induce the optimistic bound $\Delta S_k = O(K)$. However, we find their model too restrictive, since the minimum and maximum task weights can substantially deviate from $w_{avg}$. Hence, we here establish a looser and more realistic model on the task weights, so that for any subchain $\mathcal{T}_{i,j}$ with weight $W_{i,j}$ sufficiently larger than $w_{max}$, average task weight within the subchain $\mathcal{T}_{i,j}$ satisfies $\Omega(w_{avg})$. That is, $\Delta_{i,j} = j - i + 1 = O(W_{i,j}/w_{avg})$. This model, referred to here as model $\mathcal{M}$, directly induces $\Delta S_k = O(K\, w_{max}/w_{avg})$, since $\Delta W_k \le \Delta W_{K/2} = K\, w_{max}/2$ for $k = 1, 2, \ldots K-1$.

```
DP+ $(\mathcal{W}, N, K, SL, SH)$
  $B[1,i] \leftarrow \mathcal{W}[i]$ for $i = SL_1, SL_1+1, \ldots, SH_1$;
  $B[1, SH_1{+}1] \leftarrow \infty$;
  for $k \leftarrow 2$ to $K$ do
    $j \leftarrow SL_{k-1}$;
    for $i \leftarrow SL_k$ to $SH_k$ do
      if $\mathcal{W}[i] - \mathcal{W}[j] > B[k{-}1, j]$ then
        repeat $j \leftarrow j+1$ until $\mathcal{W}[i] - \mathcal{W}[j] \leq B[k{-}1, j]$;
        if $\mathcal{W}[i] - \mathcal{W}[j{-}1] < B[k{-}1, j]$ then
          $j \leftarrow j-1$;
          $B[k,i] \leftarrow \mathcal{W}[i] - \mathcal{W}[j]$;
        else
          $B[k,i] \leftarrow B[k{-}1, j]$;
      else
        $B[k,i] \leftarrow B[k-1, j]$;
    $B[1, SH_k{+}1] \leftarrow \infty$
  return $B_{opt} \leftarrow B[K, N]$;
```
Figure 6: Dynamic-programming algorithm with static seperator-index bounding.

### 4.2 Dynamic-Programming Algorithm with Static Separator-Index Bounding

The proposed DP algorithm, referred to here as the DP+ algorithm, exploits the bounds on the separator indices for the efficient solution of the CCP problem. Figure 6 illustrates the proposed DP+ algorithm where input parameters $SL$ and $SH$ denote the index bound arrays, each of size $K$, computed according to Corollary 6 with $B_f = B_{RB}$. Note that $SL_K = SH_K = N$ since only $B[K, N]$ need to be computed in the last row. As seen in Fig. 6, only $B_j^k$ values for $j = SL_k, SL_k{+}1, \ldots, SH_k$ are computed at each row $k$ by exploiting Corollary 6 which ensures the existence of an optimal partition $\Pi_{opt} = \langle s_0, s_1, \ldots, s_K\rangle$ with $SL_k \leq s_k \leq SH_k$. Thus, only these $B_j^k$ values will be sufficient for the correct computation of $B_i^{k+1}$ values for $i = SL_{k+1}, SL_{k+1}+1, \ldots, SH_{k+1}$ at the next row $k+1$.

As seen in Fig. 6, explicit range checking is not adopted in the proposed algorithm for the sake of utmost efficiency. However, the $j$-index may proceed beyond $SH_k$ to $SH_k+1$ within the *repeat-until-loop* while computing $B_i^{k+1}$ with $SL_{k+1} \leq i \leq SH_{k+1}$ in two cases. In both cases, functions $W_{j+1,i}$ and $B_j^k$ intersect in the open interval $(SH_k, SH_k+1)$ so that $B_{SH_k}^k < W_{SH_k+1,i}$ and $B_{SH_k+1}^k \geq W_{SH_k+2,i}$. In the first case, $i = SH_k+1$ so that $W_{j+1,i}$ and $B_j^k$ intersect in $(i-1, i)$, which automatically implies that $B_i^{k+1} = W_{i-1,i}$ with $j_i^{k+1} = SL_k$ since $W_{i-1,i} < B_i^k$ as mentioned earlier in Section 3.2. In the second case, $i > SH_k+1$ for which Corollary 6 guarantees that $B_i^{k+1} = W_{SL_k+1,i} \leq B_{SH_k+1}^k$ thus we can safely select $j_i^{k+1} = SL_k$. Note that $W_{SL_k+1,i} = B_{SH_k+1}^k$ may correspond to a case leading to another optimal partition with $j_i^{k+1} = s_{k+1} = SH_k+1$. As seen in Fig. 6, both cases are efficiently resolved by simply storing $\infty$ to $B_{SH_k+1}^k$ as a *sentinel*. Hence, in such cases, the condition $W_{SH_k+1,i} < B_{SH_k+1}^k = \infty$ in the *if-then* statement following the *repeat-until-loop* statement always becomes true so that the $j$-index automatically moves back to $SH_k$. Note that the scheme of computing $B_{SH_k+1}^k$ for each row $k$, which seems to be a natural solution, does not work since the correct computation of $B_{SH_{k+1}+1}^{k+1}$ may necessitate more than one $B_j^k$ value beyond the $SH_k$ index bound.

The nice property of the DP approach is that it can be used to generate all optimal partitions by maintaining a $K \times N$ matrix to store the minimum $j_i^k$ index values defining the $B_i^k$ values at the expense of increased execution time and asymptotical increase in the space requirement. Recall that the index bounds $SL$ and $SH$ computed according to Corollary 6 restrict the search space for at least one optimal solution. The index bounds can be computed according to Lemma 5 to serve that purpose, since the search space restricted by Lemma 5 includes all optimal solutions.

The running time of the proposed DP+ algorithm is $O(N+K \lg N)+\sum_{k=1}^{K} \theta(\Delta S_k)$. Here, $\theta(N)$ cost comes from the initial prefix-sum operation on the $\mathcal{W}$ array, and $O(K \lg N)$ cost comes from the running time of the RB heuristic and computing the separator-index bounds $SL$ and $SH$ according to Corollary 6. Under model $\mathcal{M}$, $\Delta S_k = O(K w_{max}/w_{avg})$, and hence the complexity is $O(N+K \lg N+K^2\, w_{max} \,/\, w_{avg})$. The algorithm becomes linear in $N$, when the separator-index ranges do not overlap, and the condition $w_{max} < 2W_{tot}/K^2$ guarantees non-overlapping index ranges.

## 4.3 Iterative Refinement Algorithms

In this work, we improve the MS algorithm and propose a novel CCP algorithm, namely the bidding algorithm, which is run-time efficient for small-to-medium number of processors. The main difference between the MS and bidding algorithms is as follows: the MS algorithm moves along a sequence of feasible bottleneck values, whereas the bidding algorithm moves along a sequence of infeasible bottleneck values such that the first feasible bottleneck value becomes the optimal.

### 4.3.1 Improving the MS Algorithm

The performance of the MS algorithm [26] strongly depends on the initial partition. The initial partition proposed by Manne and Sorevik [26] satisfies the leftist partition constraints. But it leads to very poor run-time performance. Here, we propose using the partition generated by PROBE(B*) as an initial partition. This partition is also a leftist partition, since moving any separator to the left will not help to decrease the load of the bottleneck processor. This simple observation leads to significant improvement in run-time performance of the algorithm. Also, using a heap as a priority queue does not give better run-time performance than using a running maximum despite its superior asymptotic complexity. In our implementation, we used a running maximum.

### 4.3.2 Bidding Algorithm

This algorithm increases the bottleneck value in an incremental manner, starting from the ideal bottleneck value $B^*$, until it finds a feasible partition, which happens to be an optimal one. Consider a partition $\Pi_t = \langle s_0, s_1, \ldots, s_K\rangle$ constructed by PROBE($B_t$) for an infeasible $B_t$. After detecting the infeasibility of this $B_t$ value, the important issue is to determine the next larger bottleneck value $B$ to be investigated. Clearly, the separator indices of the partitions to be constructed by the future PROBE(B) calls with $B > B_t$ will never be to the left of the respective separator indices of $\Pi_t$. Moreover, at least one of the separators should move to the right in the hope of feasibility, since load of the last processor determines the infeasibility of the current $B_t$ value (i.e., $L_K > B_t$). In order not to miss the smallest feasible bottleneck value, the next larger $B$ value is computed by selecting the minimum of the processor loads that will be obtained by moving the end-index of every processor to right by one position. That is, the next larger $B$ value is equal to $\min\{\min_{1\le k<K}\{L_k+w_{s_k+1}\}, L_K\}$. Here, we call the $L_k+w_{s_k+1}$ value as the *bid* of processor $P_k$, which refers to the load of $P_k$ if the first task $t_{s_k+1}$ of

```
BIDDING $(\mathcal{W}, N, K)$
  $s_k \leftarrow 0$ for $k \leftarrow 0, 1, \ldots, K-1$;  $s_K \leftarrow N$;
  BIDS$[0].B \leftarrow L_r \leftarrow W_{tot}$;
  $B \leftarrow B^*$;  $k \leftarrow 0$;
  while $L_r > B$ do
    repeat $k \leftarrow k+1$
      if $s_k = 0$ then
        $s_k \leftarrow$ BINSRCH $(\mathcal{W}, s_{k-1}+1, N, \mathcal{W}[s_{k-1}] + B)$;
        $L_k \leftarrow \mathcal{W}[s_k] - \mathcal{W}[s_{k-1}]$;
      else
        while $L_k + w_{s_k+1} \leq B$ do
          $s_k \leftarrow s_k + 1$;
          $L_k \leftarrow L_k + w_{s_k}$;
      $mybid \leftarrow L_k + w_{s_k+1}$;
      if $mybid \leq$ BIDS$[k-1].B$ then
        BIDS$[k].\langle B, q\rangle \leftarrow \langle mybid, k\rangle$;
      else
        BIDS$[k].\langle B, q\rangle \leftarrow$ BIDS$[k-1].\langle B, q\rangle$;
      $L_r \leftarrow W_{tot} - \mathcal{W}[s_k]$;   $rbid \leftarrow L_r \;/\; (K-k)$;
    until $rbid > B$ or $k = K-1$;
    if $rbid <$ BIDS$[k].B$ then
      $B \leftarrow rbid$;
    else
      $\langle B, k\rangle \leftarrow$BIDS$[k].\langle B, q\rangle$;
    $k \leftarrow k-1$;
  return  $B_{opt} \leftarrow B$;
```

Figure 7: Bidding algorithm.

the next processor is augmented to $P_k$. Note that the bid of the last processor $P_K$ is equal to the load of the remaining tasks. If the best bid $B$ comes from processor $P_b$, probing with new $B$ is performed only for the remaining processors $\langle P_b, P_{b+1}, \ldots P_K\rangle$ in the suffix $\mathcal{W}_{s_{b-1}+1:N}$ of the prefix-summed $\mathcal{W}$-array.

The bidding algorithm is presented in Fig. 7. The innermost *while-loop* implements a linear probing scheme, such that the new positions of the separators are determined by moving them to the right, one by one. This linear probing scheme is selected since the new positions of the separators are likely to be in a close neighborhood of the previous ones. Note that binary search is used only for setting the separator indices for the first time. After the separator index $s_k$ is set for processor $P_k$ during linear probing, the *repeat-until-loop* terminates if it is not possible to partition the remaining subchain $\mathcal{T}_{s_k+1,N}$ into $K-k$ processors without exceeding the current $B$ value, i.e., $rbid = L_r/(K-k) > B$, where $L_r$ denotes the weight of the remaining subchain. In this case, the next larger $B$ value is determined by considering the best bid among the first $k$ processors and *rbid*.

As seen in Fig. 7, we maintain a prefix-minimum array *BIDS* for computing the next larger $B$ value. Here, *BIDS* is an array of records of length $K$, where *BIDS*$[k].B$ and *BIDS*$[k].b$ store the best bid value of the first $k$ processors and the index of the processor defining it, respectively. *BIDS*$[0]$ is used to enable running prefix-minimum operation.

The complexity of the bidding algorithm for integer task weights under model $\mathcal{M}$ is $O(K \lg N + K\, w_{max} + K^2(w_{max}/w_{avg}))$. Here, $O(K \lg N)$ cost comes from the initial settings of separators through binary search. The $B$ value is increased at most $B_{opt} - B^* < w_{max}$ times, and each time the next $B$ value can be computed in $O(K)$ time, which induces the cost $O(K\, w_{max})$. The total area scanned by the separators is at most $O(K^2(w_{max}/w_{avg}))$. For non-integer task weights, the complexity can reach to $O(K \lg N + K^3(w_{max}/w_{avg}))$

```
ε-BISECT+ ($\mathcal{W}, N, SL, SH, K$)
  $LB \leftarrow B^*$;
  $UB \leftarrow B_{RB}$;
  repeat
    $B_t \leftarrow (UB + LB)$ / 2;
    if RPROBE ($B_t$) then
      for $k \leftarrow 1$ to $K-1$ do $SH_k \leftarrow s_k$;
      $UB \leftarrow B_t$;
    else
      for $k \leftarrow 1$ to $K-1$ do $SL_k \leftarrow s_k$;
      $LB \leftarrow B_t$;
  until $UB \leq LB + \epsilon$;
  return $B_{opt} \leftarrow UB$;
```

```
RPROBE ($B$)
  $Bsum \leftarrow B$;
  for $k \leftarrow 1$ to $K-1$ do
    $s_k \leftarrow$ BINSRCH ($\mathcal{W}, SL_k, SH_k, Bsum$);
    $Bsum \leftarrow \mathcal{W}[s_k] + B$;
  if $Bsum \geq W_{tot}$ then
    return TRUE;
  else
    return FALSE;
```

Figure 8: Bisection as an $\epsilon$-approximation algorithm with dynamic seperator-index bounding.

in the worst case, which occurs when only one separator index moves to the right by one position at each $B$ value. We should note here that the use of a min-heap for finding the next $B$ value enables terminating a repeat-loop iteration as soon as a separator-index does not move. The trade-off in this scheme is the $O(\lg K)$ cost to incur at each separator-index move because of the respective key-update operation on the heap. We have also implemented this scheme and experimentally verified that it increases the overall execution time in the CCP instances tested in Section 6.

## 4.4 Parametric Search Algorithms

In this work, we exploit the theoretical findings given in Section 4.1 to propose an improved probe algorithm. The improved algorithm, referred to here as the restricted probe (RPROBE), exploits the bounds computed according to Corollary 6 (with $B_f = B_{RB}$) to restrict the search space for $s_k$ separator values during the binary searches in the prefix-summed $\mathcal{W}$-array. That is, BINSRCH($\mathcal{W}, SL_k, SH_k, Bsum$) in RPROBE searches $\mathcal{W}$ in the index range $[SL_k, SH_k]$ to find the index $SL_k \leq s_k \leq SH_k$ such that $\mathcal{W}[s_k] \leq Bsum$ and $\mathcal{W}[s_k+1] > Bsum$ through binary search. This scheme reduces the complexity of an individual probe call to $\sum_{k=1}^{K} \theta(\lg \Delta_k) = O(K \lg K + K \lg(w_{max}/w_{avg}))$ by the results of Corollaries 3 and 7. Note that complexity of RPROBE reduces to $O(K \lg K)$ for sufficiently large $K$ where $K = \Omega(w_{max}/w_{avg})$. Figs. 8, 9 and 10 illustrate the RPROBE algorithms tailored for the respective parametric-search algorithms.

### 4.4.1 Approximate Bisection Algorithm with Dynamic Separator-Index Bounding

Proposed bisection algorithm, illustrated in Fig. 8, searches the space of bottleneck values in $[B^*, B_{RB}]$ range as opposed to $[B^*, W_{tot}]$. In this algorithm, if PROBE($B_t$) = TRUE, then the search space is restricted to $B \leq B_t$ values, and if PROBE($B_t$) = FALSE, then the search space is restricted to $B > B_t$ values. In this work, we exploit this simple observation to propose and develop a dynamic probing scheme to increase the efficiency of successive PROBE calls, by modifying the separator index-bounds depending on the success and the failure of the probes. Let $\Pi_t = \langle t_0, t_1, \ldots, t_K \rangle$ be the partition constructed by PROBE($B_t$). Any future PROBE($B$) call with $B \leq B_t$ will set the $s_k$ indices with $s_k \leq t_k$. Thus, the search space for $s_k$ can be restricted to those indices less than or equal to $t_k$. Similarly, any future PROBE($B$) call with $B \geq B_t$ will set the $s_k$ indices with $s_k \geq t_k$. Thus, the search space for $s_k$ can be restricted to those indices greater than or equal to $t_k$.

As illustrated in Fig. 8, dynamic update of separator-index bounds can be performed in $\theta(K)$ time through a

```
EXACT-BISECT ($\mathcal{W}, N, SL, SH, K$)
  $LB \leftarrow B^*$;   $UB \leftarrow B_{RD}$;
  repeat
    $B_t \leftarrow (UB + LB)\ /\ 2$;
    if RPROBE ($B_t$) then
      for $k \leftarrow 1$ to $K-1$ do $SH_k \leftarrow s_k$;
      $UB \leftarrow \max_{1\leq k\leq K}\{L_k\}$;
    else
      for $k \leftarrow 1$ to $K-1$ do $SH_k \leftarrow s_k$;
      $LB \leftarrow \min\{\min_{1\leq k<K}\{L_k + w_{s_k+1}\}, L_K\}$
  until $UB = LB$;
  return $B_{opt} \leftarrow UB$;
```

```
RPROBE ($B$)
  $Bsum \leftarrow B$;
  for $k \leftarrow 1$ to $K-1$ do
    $s_k \leftarrow$ BINSRCH ($\mathcal{W}, SL_k, SH_k, Bsum$);
    $Bsum \leftarrow \mathcal{W}[s_k] + B$;
    $L_k \leftarrow \mathcal{W}[s_k] - \mathcal{W}[s_{k-1}]$
  $L_K \leftarrow \mathcal{W}[N] - \mathcal{W}[s_{K-1}]$
  if $L_K \leq B$ then
    return TRUE
  else
    return FALSE
```

Figure 9: Exact bisection algorithm with dynamic seperator-index bounding.

simple *for-loop* over $SL$ or $SH$ arrays depending on failure or success of RPROBE($B_t$), respectively. However, in our implementation, this update is efficiently achieved in $O(1)$ time through the pointer assignment $SL \leftarrow \Pi$ or $SH \leftarrow \Pi$ depending on failure or success of the RPROBE($B_t$).

Similar to the $\epsilon$-BISECT algorithm, the proposed $\epsilon$-BISECT+ algorithm is also an $\epsilon$-approximation algorithm for general workload arrays. However, both $\epsilon$-BISECT and $\epsilon$-BISECT+ algorithms become exact algorithm for integer-valued workload arrays by setting $\epsilon = 1$. As shown in Lemma 1, $B_{RB} < B^* + w_{max}$. Hence, for integer-valued workload arrays the maximum number of probe calls in the $\epsilon$-BISECT+ algorithm is lg $w_{max}$, thus the overall complexity is $O(N + K \lg N + \lg(w_{max})(K \lg K + K \lg(w_{max}/w_{avg})))$ under model $\mathcal{M}$. Here, $\theta(N)$ cost comes from the initial prefix-sum operation on the $\mathcal{W}$ array, and $O(K \lg N)$ cost comes from the running time of the RB heuristic and computing the separator-index bounds $SL$ and $SH$ according to Corollary 6.

### 4.4.2 Bisection as an Exact Algorithm

In this work, we enhance the bisection algorithm to be an exact algorithm for general workload arrays by appropriate update of the lower and upper bounds after each probe call. The idea is to move the upper and lower bounds on the value of an optimal solution to a realizable bottleneck value, i.e., the total weight of a subchain of $\mathcal{W}$ after each probe call. This reduces the search space to the finite set of realizable bottleneck values, as opposed to the infinite space of bottleneck values defined by a range $[LB, UB]$. Each bisection step is designed to eliminate at least one candidate value, thus the algorithm terminates in finite number of steps to find the optimal bottleneck value.

After a probe call RPROBE($B_t$), current upper bound value $UB$ is modified if RPROBE($B_t$) succeeds. Note that RPROBE($B_t$) not only determines the feasibility of $B_t$, but also constructs a partition $\Pi$ with $cost(\Pi_t) \leq B_t$. Instead of reducing the upper bound $UB$ to $B_t$ we can further reduce $UB$ to the bottleneck value $B = cost(\Pi_t) \leq B_t$ of the partition $\Pi_t$ constructed by RPROBE($B_t$). Similarly, current lower bound $LB$ is modified when RPROBE($B_t$) fails. In this case, instead of increasing the lower bound $LB$ to $B_t$, we can exploit the partition $\Pi_t$ constructed by RPROBE($B_t$) to increase $LB$ further to the smallest realizable bottleneck value $B$ greater than $B_t$. How to compute this value is already described in our bidding algorithm:

$$B = \min\{\min_{1\leq k<K}\{L_k + w_{s_k+1}\}, L_K\}$$

where $L_k$ denotes the load of processor $P_k$ in $\Pi_t$. Fig. 9 presents the pseudocode of our algorithm.

Each bisection step divides the set of candidate realizable bottleneck values into two sets, and eliminates one. The initial set can have a size between 1 and $N^2$, initial number of realizable bottleneck values. Assuming that they are equally likely (a more even distribution is likely if $B_t$ is chosen as the medium of $LB$ and $UB$), the expected complexity of the algorithm will be

$$T(N) = \frac{1}{N^2}\sum_{i=1}^{N^2} T(i) + O(K \lg K + K \lg(w_{max}/w_{avg})),$$

which has the solution $O(K \lg K \lg N + K \lg N \lg(w_{max}/w_{avg}))$. Here, $O(K \lg K + K \lg(w_{max}/w_{avg}))$ is the cost of a probe operation, and $\lg N$ is the expected number of probe calls. Thus, the overall complexity becomes $O(N + K \lg K \lg N + K \lg N \lg(w_{max}/w_{avg}))$, where $\theta(N)$ cost comes from the initial prefix-sum operation on the $\mathcal{W}$ array.

### 4.4.3 Improving Nicol's Algorithm as a Divide-and-Conquer Algorithm

The theoretical findings presented in previous sections can be exploited at different levels to introduce improvement to the performance of Nicol's algorithm. The obvious improvement is to use the proposed restricted probe function instead of the conventional probe functions. The careful implementation scheme given in Fig. 5(b) enables the use of dynamic separator-index bounding. In this work, we exploit the idea behind the bisection algorithm to propose an efficient divide-and-conquer approach for Nicol's algorithm for further improvement.

Consider the sequence of probes of the form PROBE($W_{1,j}$) performed by Nicol's algorithm for processor $P_1$ to find the smallest index $j = i_1$ such that PROBE($W_{1,j}$) = TRUE. Starting from a naive bottleneck-value range ($LB_0 = 0,\ UB_0 = W_{tot}$), the success and failure of these probe calls can be exploited to narrow this range to $(LB_1, UB_1)$. That is, each PROBE($W_{1,j}$) = TRUE decreases the upper bound to $W_{1,j}$ and each PROBE($W_{1,j}$) =FALSE increases the lower bound to $W_{1,j}$. It is clear that we will have $(LB_1 = W_{1,i_1-1},\ UB_1 = W_{1,i_1})$ at the end of this search process for processor $P_1$. Now, consider the sequence of probes of the form PROBE($W_{i_1,j}$) performed for processor $P_2$ to find the smallest index $j = i_2$ such that PROBE($W_{i_1,j}$) = TRUE. Our key observation is that the partition $\Pi_t = \langle 0, t_1, t_2, \ldots, t_{K-1}, N\rangle$ to be constructed by any PROBE($B_t$) with $LB_1 < B_t = W_{i_1,j} < UB_1$ will satisfy $t_1 = i_1 - 1$ since $W_{1,i_1-1} < B_t < W_{1,i_1}$. Hence, probe calls with $LB_1 < B_t < UB_1$ for processor $P_2$ can be restricted to be performed in $\mathcal{W}_{i_1:N}$, where $\mathcal{W}_{i_1:N}$ denotes the $(N - i_1 + 1)$th suffix of the prefix-summed $\mathcal{W}$ array. This simple yet effective scheme leads to an efficient divide-and-conquer algorithm as follows. Let $\mathcal{T}_i^{K-k}$ denote the CCP subproblem of $(K-k)$-way partitioning of the $(N-i+1)$th suffix $\mathcal{T}_{i,N} = \langle t_i, t_{i+1}, \ldots, t_N\rangle$ of the task chain $\mathcal{T}$ onto the $(K-k)$th suffix $\mathcal{P}_{k,K} = \langle P_{k+1}, P_{k+2}, \ldots, P_K\rangle$ of the processor chain $\mathcal{P}$. Once the index $i_1$ for processor $P_1$ is computed, the optimal bottleneck value $B_{opt}$ can be defined by either $W_{1,i_1}$ or the the bottleneck value of an optimal $(K-1)$-way partitioning of the suffix subchain $\mathcal{T}_{i_1,N}$. That is, $B_{opt} = \min\{B_1 = W_{1,i_1}, C(\Pi_{i_1}^{K-1})\}$. Proceeding this way, once the indices $\langle i_1, i_2, \ldots, i_k\rangle$ for the first $k$ processors $\langle P_1, P_2, \ldots, P_k\rangle$ are determined, $B_{opt} = \min\{\min_{1\le b\le k}\{B_b = W_{i_{b-1},i_b}\}, C(\Pi_{i_k}^{K-k})\}$.

This divide-and-conquer approach is presented in Fig. 10. At the $b$th iteration of the outer *for-loop*, given $i_{b-1}$, $i_b$ is found in the inner *while-loop* by conducting probes on $\mathcal{W}_{i_{b-1}:N}$ to compute $B_b = W_{i_{b-1},i_b}$. As seen in Fig. 10, the dynamic bounds on the separator indices are exploited in two distinct ways according to Theorem 8. First, the restricted probe function RPROBE is used for probing. Second, the search space for the bottleneck

```
NICOL+ $(\mathcal{W}, N, SL, SH, K)$
  $i_0 \leftarrow 1$;  $LB \leftarrow W_{tot}/K$;  $UB \leftarrow B_{RB}$;
  for $b \leftarrow 1$ to $K-1$ do
    $ilow \leftarrow SL_b$;  $ihigh \leftarrow SH_b$;
    while $ilow < ihigh$ do
      $imid \leftarrow (ilow + ihigh)\ /\ 2$;
      $B \leftarrow \mathcal{W}[imid] - \mathcal{W}[i_{b-1} - 1]$;
      if $LB \leq B < UB$ then
        if RPROBE $(b, imid, B)$ then
          for $k \leftarrow b+1$ to $K-1$ do  $SH_k \leftarrow s_k$;
          $UB \leftarrow B$;
          $ihigh \leftarrow imid$;
        else
          for $k \leftarrow b+1$ to $K-1$ do  $SL_k \leftarrow s_k$;
          $LB \leftarrow B$;
          $ilow \leftarrow imid + 1$;
      elseif $B \geq UB$
        $ihigh \leftarrow imid$;
      else
        $ilow \leftarrow imid + 1$;
    $i_b \leftarrow ihigh$;
    $B_b \leftarrow \mathcal{W}[i_b] - \mathcal{W}[i_{b-1} - 1]$;
  $B_K \leftarrow \mathcal{W}[N] - W[i_{K-1} - 1]$;
  return $\min_{1 \leq b \leq K}\{B_b\}$;
```

```
RPROBE $(b, imid, B)$
  $Bsum \leftarrow \mathcal{W}[imid] + B$;
  for $k \leftarrow b+1$ to $K-1$ do
    $s_k \leftarrow$BINSRCH $(\mathcal{W}, SL_k, SH_k, Bsum)$;
    $Bsum \leftarrow W[s_k] + B$;
  if $Bsum \geq W_{tot}$ then
    return TRUE;
  else
    return FALSE;
```

Figure 10: Nicol's algorithm with dynamic separator-index bounding.

values of the processors is restricted. That is, given $i_{b-1}$, the binary search for $i_b$ over all subchain weights of the form $W_{i_{b-1}+1,j}$ for $i_{b-1} < j \leq N$ is restricted to $W_{i_{b-1}+1,j}$ values for $SL_b \leq j \leq SH_b$.

Under model $\mathcal{M}$, the complexity of this algorithm is $O(N + K \lg N + w_{max}(K \lg K + K \lg(w_{max}/w_{avg})))$ for integer task weights. Because, the number of probe calls cannot exceed $w_{max}$, since there are at most $w_{max}$ distinct bound values in the range $[B^*, B_{RB}]$. For non-integer task weights, the complexity can be given as $O(N + K \lg N + w_{max}(K \lg K)^2 + w_{max}K^2 \lg K \lg(w_{max}/w_{avg}))$, since the algorithm makes $O(w_{max} K \lg K)$ probe calls. Here, $\theta(N)$ cost comes from the initial prefix-sum operation on the $\mathcal{W}$ array, and $O(K \lg N)$ cost comes from the running time of the RB heuristic and computing the separator-index bounds $SL$ and $SH$ according to Corollary 6.

## 5 Load Balancing Applications

Here, we describe load-balancing applications used to test performance of proposed CCP algorithms.

### 5.1 Parallel Sparse Matrix Vector Multiplication

Sparse matrix vector multiplication (SpMxV) is one of the most important kernels in scientific computing. Parallelization of repeated SpMxV computations requires partitioning and distribution of the sparse matrix. Two possible 1D sparse-matrix partitioning schemes are *rowwise striping* (RS) and *columnwise striping* (CS). Consider parallelization of SpMxV operations of the form $y = A\,x$ in an iterative solver, where $A$ is an $N \times N$ sparse matrix, and $y$ and $x$ are $N \times 1$ vectors. In RS, processor $P_k$ owns the $k$th row stripe $A^r_k$ of $A$ and it is responsible for computing $y_k = A^r_k\,x$, where $y_k$ denotes the $k$th stripe of vector $y$. In CS, processor $P_k$ owns the $k$th column stripe $A^c_k$ of $A$ and is responsible for computing $y^k = A^c_k\,x$, where $y = \sum_{k=1}^{K} y^k$. All vectors used in the solver are divided conformally with row or column partitioning in the RS or CS schemes, respectively, in order to avoid the communication of vector components during the linear vector operations. The RS and CS schemes require communication before or after the local SpMxV computations, thus they can also be considered as *pre* and *post* communication schemes, respectively. In RS, each task $t_i \in \mathcal{T}$ corresponds to the atomic task of computing the inner-product of row $i$ of matrix $A$ with the column vector $x$. In CS, each task $t_i \in \mathcal{T}$ corresponds to the atomic task of computing the sparse DAXPY operation $y = y + x_i a_{*i}$, where $a_{*i}$ denotes the $i$th column of $A$. Hence, each nonzero entry in a row and column of A incurs a multiply-and-add operation during the local SpMxV computations. Thus, computational load $w_i$ of task $t_i$ is the number of nonzero entries in row $i$ (column $i$) in the RS (CS) scheme. So, the load balancing problem in the rowwise and columnwise block partitioning of a sparse matrix in the given ordering can be modeled as the CCP problem.

In RS (CS), by allowing only row (column) reordering, the load balancing problem can be described as the number partitioning problem, which is known to be NP-Hard [10]. By allowing both row and column reordering, the problem of minimizing communication overhead while maintaining load balance can be described as graph and hypergraph partitioning problems [5, 14], which are also known to be NP-Hard [9, 25]. However, possibly high preprocessing overhead involved in these models may not be justified in some applications. If the partitioner is to be used as a run-time library in a parallelizing compiler for a data-parallel programming language [39, 41], row and column reordering lead to high memory requirement due to the irregular mapping table and extra level of indirection in locating distributed data during each multiply-and-add operation [40]. Furthermore, in some applications, the natural row and column ordering of the sparse matrix may already be likely to induce small communication overhead (e.g., banded matrices).

The proposed CCP algorithms are surprisingly fast such that the initial prefix-sum operation dominates their execution times in sparse-matrix partitioning. In this work, we exploit compressed storage schemes of sparse matrices to avoid the initial prefix-sum operation as follows: We assume the use of *compressed row storage* (CRS) and *compressed column storage* (CCS) schemes for rowwise and columnwise striping, respectively. In CRS, an array DATA of length NZ stores nonzeros of matrix $A$, in row-major order, where NZ $= W_{tot}$ denotes the total number of nonzeros in $A$. An index array COL of length NZ stores the column indices of the respective nonzeros in array DATA. Another index array ROW of length $N+1$ stores the starting indices of the respective rows in the other two arrays. Hence, any subchain weight $W_{i,j}$ can be efficiently computed using $W_{i,j} = \mathrm{ROW}[j+1] - \mathrm{ROW}[i]$ in $O(1)$ time without any preprocessing overhead. CCS is similar to CRS with rows and columns interchanged, thus $W_{i,j}$ is computed using $W_{i,j} = \mathrm{COL}[j+1] - \mathrm{COL}[i]$ in $O(1)$ time.

## 5.2 Sort-First Parallel Direct Volume Rendering

Direct volume rendering (DVR) methods are widely used in rendering unstructured volumetric grids for visualization and interpretation of computer simulations performed for investigating physical phenomena in various fields of science and engineering. A DVR application contains two interacting domains: object space and image space. Object space is a 3D domain containing the volume data to be visualized. Image space (screen) is a 2D domain containing pixels from which rays are shot into the 3D object domain to determine the color values of the respective pixels. Based on these domains, there are basically two approaches for data parallel DVR: image-space and object-space parallelism, which are also called as sort-first and sort-last parallelism according to the taxonomy based on the point of data redistribution point in the rendering pipeline [28]. Pixels or pixel blocks constitute the atomic tasks in sort-first parallelism, whereas volume elements (primitives) constitute the atomic tasks in sort-last parallelism.

In sort-first parallel DVR, screen is decomposed into regions and each region is assigned to a separate processor for local rendering. The primitives, whose projection areas intersect more than one region, are replicated in the processors assigned to those regions. Sort-first parallelism is a promising approach since each processor generates a complete image for its local screen subregion. However, it faces load-balancing problems in the DVR of unstructured grids due to uneven on-screen primitive distribution.

Image-space decomposition schemes for sort-first parallel DVR can be classified as *static* and *adaptive* [24]. Static decomposition is a view-independent scheme and the load-balancing problem is solved implicitly by scattered assignment of pixels or pixel blocks. Load-balancing performance of this scheme depends on the assumption that neighbor pixels are likely to have equal workload since they are likely to have similar views of the volume. As the scattered assignment scheme assigns adjacent pixels or pixel blocks to different processors, it disturbs image-space coherency and increases the amount of primitive replication. Adaptive decomposition is a view-dependent scheme and load-balancing problem is solved explicitly by using the primitive distribution on the screen.

In adaptive image-space decomposition, the number of primitives with bounding-box approximation is taken to be the workload of a screen region. Primitives constituting the volume are tallied to a 2D coarse mesh superimposed on the screen. Some primitives may intersect multiple cells. The inverse-area heuristic [29] is used to decrease the amount of errors due to counting such primitives many times. Each primitive increments the weight of each cell it intersects by a value inversely proportional to the number of cells the primitive intersects. In this heuristic, if we assume that there are no shared primitives among screen regions, the sum of the weights of individual mesh cells forming a region gives the number of primitives in that region. However, shared primitives may still cause some errors, but such errors are much less than counting such primitives multiple times while adding mesh-cell weights.

Minimizing the perimeter of the resulting regions in the decomposition is expected to minimize the communication overhead due to the shared primitives. 1D decomposition, i.e., horizontal and vertical striping of the screen, suffer from unscalability. Hilbert space-filling curve [35] is widely used for 2D decomposition of 2D non-uniform workloads. In this scheme [24], the 2D coarse mesh superimposed on the screen is traversed according to the Hilbert curve to map the 2D coarse mesh to a 1D chain of mesh cells. Load-balancing problem in this decomposition scheme reduces to the CCP problem. Using Hilbert curve as the space-filling curve is an implicit effort towards reducing the total perimeter since Hilbert curve avoids jumps during the traversal of the 2D coarse mesh. Note that 1D workload array to be used for partitioning is a real-valued array because of the inverse-area heuristic used for computing the weights of the coarse-mesh cells.

Table 1: Properties of the test set.

| SPARSE-MATRIX DATASET | | | | | | | | DIRECT VOLUME RENDERING (DVR) DATASET | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| name | # of tasks | workload: # of nonzeros | | | | ex. time SpMxV | | name | # of tasks | workload | | | |
| | | total | per row/col (task) | | | | | | | total | per task | | |
| | $N$ | $W_{tot}$ | $w_{avg}$ | $w_{min}$ | $w_{max}$ | msecs | | | $N$ | $W_{tot}$ | $w_{avg}$ | $w_{min}$ | $w_{max}$ |
| NL | 7039 | 105089 | 14.93 | 1 | 361 | 22.55 | | blunt256 | 17303 | 303K | 17.54 | 0.020 | 1590.97 |
| cre-d | 8926 | 372266 | 41.71 | 1 | 845 | 72.20 | | blunt512 | 93231 | 314K | 3.36 | 0.004 | 661.50 |
| CQ9 | 9278 | 221590 | 23.88 | 1 | 702 | 45.90 | | blunt1024 | 372824 | 352K | 0.94 | 0.001 | 411.04 |
| ken-11 | 14694 | 82454 | 5.61 | 2 | 243 | 19.65 | | post256 | 19653 | 495K | 25.19 | 0.077 | 3245.50 |
| mod2 | 34774 | 604910 | 17.40 | 1 | 941 | 124.05 | | post512 | 134950 | 569K | 4.22 | 0.015 | 1092.00 |
| world | 34506 | 582064 | 16.87 | 1 | 972 | 119.45 | | post1024 | 539994 | 802K | 1.49 | 0.004 | 1546.78 |

## 6 Experimental Results

All CCP algorithms were implemented in *C* language. All experiments were carried out on a workstation equipped with a 133MHz *PowerPC* and 64 MB of memory. We have experimented $K = 16, 32, 64, 128, 256$ way partitioning of each test data.

The dataset for sparse-matrix decomposition comes from sparse test matrices arising in linear programming problems obtained from *Netlib* suite [11] and *IOWA Optimization Center* [15]. The sparsity pattern of these matrices are obtained by multiplying the respective rectangular constraint matrices with their transposes. Hence, load balancing in the rowwise and columnwise decompositions are equivalent since the resulting matrices are symmetric. Table 1 illustrates the properties of the test matrices. Note that the number of tasks in the sparse-matrix (SpM) dataset also refers to the number of rows and columns of the respective matrix. The execution time of a single SpMxV operation for each test matrix is also displayed in this table.

The dataset for image-space decomposition comes from sort-first parallel DVR of curvilinear grids *blunt-fin* and *post* representing the results of computational fluid dynamic simulations. These grids are commonly used by researchers in the volume rendering field. Raw grids consist of hexahedral elements and are converted into unstructured tetrahedral data format by dividing each hexahedron into 5 tetrahedrons. Triangular faces of tetrahedrons constitute the primitives mentioned in Section 5.2. Three distinct 1D workload arrays are constructed both for *blunt-fin* and *post* as described in Section 5.2 for coarse meshes of resolutions 256$\times$256, 512$\times$512, and 1024$\times$1024 superimposed on the screen of resolution 1024$\times$1024. Properties of these six workload arrays are displayed in Table 1. The number of tasks is much less than the coarse-mesh resolution, because of the zero-weight tasks in the 1D workload arrays. The CCP problem on such workload arrays can be solved by compressing tasks with nonzero weights. In Table 1, the $w_{avg}$, $w_{min}$ and $w_{max}$ columns display the average, minimum and maximum task weights for both datasets.

Following abbreviations are used for the CCP algorithms: H1 and H2 refer to Miguet and Pierson's [27] heuristics described in Section 3.1. RB refers to the recursive-bisection heuristic described in Section 3.1. DP refers to the $O((N-K)K)$-time dynamic-programming algorithm given in Fig. 1. MS refers to Manne and Sorevik's [26] iterative-refinement algorithm given in Fig. 2. $\epsilon$BS refers to the $\epsilon$-approximate parametric-search-based bisection algorithm given in Fig. 4. NC- and NC refer to the straightforward and careful implementations of Nicol's [33] parametric-search algorithm given in Fig. 5(a) and Fig. 5(b), respectively. Abbreviations ending with "+" are used to represent our improved versions of the above algorithms. That is, DP+, $\epsilon$BS+, NC+ refer to our algorithms given in Fig. 6, Fig. 8, Fig. 10, respectively, and MS+ refers to the algorithm described in

Table 2: Percent load imbalance values.

| SPARSE-MATRIX DATASET | | | | | | | DVR DATASET | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CCP instance | | HEURISTICS | | | OPT | | CCP instance | | HEURISTICS | | | OPT |
| name | K | H1 | H2 | RB | | | name | K | H1 | H2 | RB | |
| | 16 | 2.60 | 2.44 | 1.20 | 0.35 | | | 16 | 4.60 | 1.20 | 0.49 | 0.34 |
| | 32 | 5.02 | 5.75 | 3.44 | 0.95 | | | 32 | 6.93 | 2.61 | 1.94 | 1.12 |
| NL | 64 | 8.95 | 9.01 | 5.60 | 2.37 | | blunt256 | 64 | 14.52 | 9.44 | 9.44 | 2.31 |
| | 128 | 33.13 | 27.16 | 22.78 | 4.99 | | | 128 | 38.25 | 24.39 | 16.67 | 4.82 |
| | 256 | 69.55 | 69.55 | 60.78 | 14.25 | | | 256 | 96.72 | 37.03 | 34.21 | 34.21 |
| | 16 | 2.27 | 0.98 | 0.53 | 0.45 | | | 16 | 0.95 | 0.98 | 0.98 | 0.16 |
| | 32 | 4.19 | 4.42 | 3.74 | 0.63 | | | 32 | 1.38 | 1.38 | 1.18 | 0.33 |
| cre-d | 64 | 7.12 | 4.92 | 4.34 | 1.73 | | blunt512 | 64 | 2.87 | 2.69 | 1.66 | 0.53 |
| | 128 | 25.57 | 18.73 | 16.70 | 2.88 | | | 128 | 5.62 | 8.45 | 4.62 | 0.97 |
| | 256 | 37.54 | 26.81 | 35.20 | 10.95 | | | 256 | 14.18 | 14.33 | 9.34 | 2.28 |
| | 16 | 1.85 | 1.85 | 0.58 | 0.58 | | | 16 | 0.94 | 0.57 | 0.95 | 0.10 |
| | 32 | 5.65 | 2.88 | 2.24 | 0.90 | | | 32 | 1.89 | 1.21 | 0.97 | 0.14 |
| CQ9 | 64 | 13.25 | 11.49 | 7.64 | 1.43 | | blunt1024 | 64 | 4.99 | 2.16 | 1.44 | 0.26 |
| | 128 | 33.96 | 32.22 | 22.34 | 3.51 | | | 128 | 10.06 | 4.25 | 2.47 | 0.57 |
| | 256 | 58.62 | 58.62 | 58.62 | 14.72 | | | 256 | 19.65 | 9.68 | 9.68 | 0.98 |
| | 16 | 3.74 | 2.01 | 0.98 | 0.21 | | | 16 | 1.10 | 1.43 | 0.76 | 0.56 |
| | 32 | 3.74 | 4.67 | 3.74 | 1.18 | | | 32 | 3.23 | 3.98 | 3.23 | 1.11 |
| ken-11 | 64 | 13.17 | 13.17 | 13.17 | 1.29 | | post256 | 64 | 17.28 | 11.04 | 10.90 | 3.10 |
| | 128 | 13.17 | 16.89 | 13.17 | 6.80 | | | 128 | 45.35 | 29.09 | 29.09 | 8.29 |
| | 256 | 50.99 | 50.99 | 50.99 | 7.11 | | | 256 | 67.86 | 67.86 | 67.86 | 67.86 |
| | 16 | 0.06 | 0.06 | 0.06 | 0.03 | | | 16 | 0.49 | 1.25 | 0.33 | 0.33 |
| | 32 | 0.19 | 0.19 | 0.19 | 0.07 | | | 32 | 0.94 | 1.61 | 0.90 | 0.58 |
| mod2 | 64 | 7.42 | 2.72 | 2.18 | 0.18 | | post512 | 64 | 4.85 | 5.33 | 4.85 | 0.94 |
| | 128 | 16.15 | 6.29 | 2.46 | 0.41 | | | 128 | 18.03 | 14.55 | 10.15 | 1.72 |
| | 256 | 19.47 | 19.47 | 18.92 | 1.23 | | | 256 | 30.03 | 25.29 | 25.29 | 3.73 |
| | 16 | 0.27 | 0.18 | 0.09 | 0.04 | | | 16 | 0.70 | 0.70 | 0.53 | 0.20 |
| | 32 | 0.63 | 0.37 | 0.27 | 0.08 | | | 32 | 1.49 | 1.49 | 1.41 | 0.54 |
| world | 64 | 4.73 | 4.73 | 4.73 | 0.28 | | post1024 | 64 | 2.85 | 2.85 | 1.49 | 0.91 |
| | 128 | 6.37 | 6.37 | 6.37 | 0.76 | | | 128 | 13.15 | 11.79 | 9.10 | 1.11 |
| | 256 | 27.99 | 27.41 | 27.41 | 1.11 | | | 256 | 40.50 | 13.37 | 14.19 | 2.54 |
| AVERAGES OVER K | | | | | | | | | | | | |
| | 16 | 1.76 | 1.15 | 0.74 | 0.36 | | | 16 | 1.44 | 1.01 | 0.64 | 0.28 |
| | 32 | 3.99 | 3.39 | 2.88 | 0.76 | | | 32 | 2.64 | 2.05 | 1.61 | 0.64 |
| | 64 | 9.01 | 6.78 | 5.69 | 1.43 | | | 64 | 7.89 | 5.58 | 4.96 | 1.31 |
| | 128 | 19.97 | 17.66 | 14.09 | 3.36 | | | 128 | 21.74 | 15.42 | 12.02 | 2.91 |
| | 256 | 43.89 | 38.91 | 36.04 | 9.18 | | | 256 | 44.82 | 27.93 | 26.76 | 18.60 |

Section 4.3.1. BID refers to our new iterative-refinement-based bidding algorithm given in Fig. 7. EBS refers to our exact bisection algorithm given in Fig. 9. Both $\epsilon$BS and $\epsilon$BS+ algorithms are effectively used as exact algorithms for the SpM dataset with $\epsilon = 1$ by exploiting the integer-valued workload arrays arising in the SpM dataset. However, performance of these two algorithms are not tested on the DVR dataset, since they remain to be approximation algorithms because of the real-valued task weights in DVR.

Table 2 compares load-balancing quality of the heuristics and exact algorithms. In this table, percent load-imbalance values are computed as $100 \times (B_{max} - B^*)/B^*$, where $B_{max}$ denotes the bottleneck value of the respective partition and $B^* = W_{tot}/K$ denotes the ideal bottleneck value. OPT values refer to the load-imbalance values of the optimal partitions produced by exact algorithms, i.e., $B_{max} = B_{opt}$. Table 2 clearly shows that considerably better partitions can be obtained in both SpM and DVR datasets by using exact load-balancing algorithms instead of heuristics. The quality gap between the solutions of exact algorithms and heuristics increases with increasing $K$. Only exceptions to these observations are 256-way partitioning of DVR instances *blunt256* and *post256*, for which the RB heuristic achieves to obtain optimal partitions.

Table 3 compares performances of the static separator-index bounding schemes discussed in Section 4.1. The values displayed in this table correspond to the sum of the sizes of the separator-index ranges normalized with respect to $N$, i.e., $\sum_{k=1}^{K-1} \Delta S_k/N$, where $\Delta S_k = SH_k - SL_k + 1$. For each CCP instance, $(N-K)K$

Table 3: Sizes of separator-index ranges normalized with respect to $N$.

| SPARSE-MATRIX DATASET | | | | | | DVR DATASET | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| CCP instance | | | | | | CCP instance | | | | | |
| name | K | (N-K)K | L2 | L4 | C6 | name | K | (N-K)K | L2 | L4 | C6 |
| | 16 | 15.97 | 0.32 | 0.13 | 0.036 | | 16 | 15.99 | 0.38 | 0.07 | 0.003 |
| | 32 | 31.86 | 1.23 | 0.58 | 0.198 | | 32 | 31.94 | 1.22 | 0.30 | 0.144 |
| NL | 64 | 63.43 | 4.97 | 2.24 | 0.964 | blunt256 | 64 | 63.77 | 5.10 | 3.81 | 0.791 |
| | 128 | 125.69 | 19.81 | 19.60 | 4.305 | | 128 | 127.06 | 21.66 | 12.88 | 2.784 |
| | 256 | 246.73 | 77.96 | 82.83 | 22.942 | | 256 | 252.23 | 101.68 | 50.48 | 10.098 |
| | 16 | 15.97 | 0.16 | 0.04 | 0.019 | | 16 | 16.00 | 0.09 | 0.17 | 0.006 |
| | 32 | 31.89 | 0.76 | 0.90 | 0.137 | | 32 | 31.99 | 0.35 | 0.26 | 0.043 |
| cre-d | 64 | 63.55 | 2.89 | 1.85 | 0.610 | blunt512 | 64 | 63.96 | 1.64 | 0.69 | 0.144 |
| | 128 | 126.18 | 11.59 | 15.12 | 2.833 | | 128 | 127.83 | 6.65 | 4.59 | 0.571 |
| | 256 | 248.69 | 46.66 | 57.54 | 16.510 | | 256 | 255.30 | 28.05 | 16.71 | 2.262 |
| | 16 | 15.97 | 0.30 | 0.03 | 0.023 | | 16 | 16.00 | 0.05 | 0.09 | 0.010 |
| | 32 | 31.89 | 1.21 | 0.34 | 0.187 | | 32 | 32.00 | 0.18 | 0.18 | 0.018 |
| CQ9 | 64 | 63.57 | 4.84 | 2.96 | 0.737 | blunt1024 | 64 | 63.99 | 0.77 | 0.74 | 0.068 |
| | 128 | 126.25 | 19.20 | 18.66 | 3.121 | | 128 | 127.96 | 3.43 | 2.53 | 0.300 |
| | 256 | 248.96 | 74.84 | 86.80 | 23.432 | | 256 | 255.82 | 13.92 | 20.36 | 1.648 |
| | 16 | 15.98 | 0.28 | 0.11 | 0.024 | | 16 | 15.99 | 0.35 | 0.04 | 0.021 |
| | 32 | 31.93 | 1.10 | 1.13 | 0.262 | | 32 | 31.95 | 1.50 | 0.52 | 0.162 |
| ken-11 | 64 | 63.73 | 4.42 | 7.34 | 0.655 | post256 | 64 | 63.79 | 6.12 | 4.26 | 0.932 |
| | 128 | 126.89 | 17.60 | 14.56 | 4.865 | | 128 | 127.17 | 26.48 | 23.68 | 4.279 |
| | 256 | 251.56 | 69.18 | 85.44 | 13.076 | | 256 | 252.68 | 124.76 | 90.48 | 16.485 |
| | 16 | 15.99 | 0.16 | 0.00 | 0.002 | | 16 | 16.00 | 0.13 | 0.02 | 0.003 |
| | 32 | 31.97 | 0.55 | 0.04 | 0.013 | | 32 | 31.99 | 0.56 | 0.14 | 0.063 |
| mod2 | 64 | 63.88 | 2.14 | 1.22 | 0.109 | post512 | 64 | 63.97 | 2.33 | 2.41 | 0.413 |
| | 128 | 127.53 | 8.62 | 2.59 | 0.411 | | 128 | 127.88 | 9.33 | 10.15 | 1.714 |
| | 256 | 254.12 | 34.49 | 39.02 | 2.938 | | 256 | 255.52 | 37.08 | 46.28 | 6.515 |
| | 16 | 15.99 | 0.15 | 0.01 | 0.004 | | 16 | 16.00 | 0.17 | 0.07 | 0.020 |
| | 32 | 31.97 | 0.59 | 0.06 | 0.020 | | 32 | 32.00 | 0.54 | 0.27 | 0.087 |
| world | 64 | 63.88 | 2.30 | 2.81 | 0.158 | post1024 | 64 | 63.99 | 2.19 | 0.61 | 0.210 |
| | 128 | 127.53 | 9.23 | 7.19 | 0.850 | | 128 | 127.97 | 8.77 | 9.48 | 0.918 |
| | 256 | 254.11 | 36.97 | 53.15 | 2.635 | | 256 | 255.88 | 34.97 | 27.28 | 4.479 |
| AVERAGES OVER K | | | | | | | | | | | |
| | 16 | 15.97 | 0.21 | 0.06 | 0.018 | | 16 | 16.00 | 0.20 | 0.08 | 0.011 |
| | 32 | 31.88 | 0.86 | 0.68 | 0.136 | | 32 | 31.98 | 0.73 | 0.28 | 0.096 |
| | 64 | 63.52 | 3.43 | 2.63 | 0.516 | | 64 | 63.91 | 3.03 | 2.09 | 0.426 |
| | 128 | 126.07 | 13.68 | 12.74 | 2.731 | | 128 | 127.65 | 12.72 | 10.55 | 1.428 |
| | 256 | 248.26 | 54.28 | 57.55 | 14.089 | | 256 | 254.57 | 56.74 | 41.93 | 6.915 |

*L2, L4 and C6 denote separator-index ranges obtained according to results of Lemma 2, Lemma 4 and Corollary 6.*

represents the size of the search space for the separator indices and the total number of table entries referenced and computed by the DP algorithm. Columns labeled as L2, L4 and C6 display the total range sizes obtained according to the results of Lemma 2, Lemma 4 and Corollary 6, respectively.

As seen in Table 3, proposed practical scheme C6 achieves substantially better separator-index bounds than L2, despite its worse worst-case behavior (see Corollaries 3 and 7). Comparison of columns L4 and C6 displays the substantial benefit of performing left-to-right and right-to-left probes with $B^*$ according to Lemma 5. Comparison of $(N-K)K$ and C6 columns shows that the proposed separator-index bounding scheme is very effective in restricting the search space for the separator indices in both SpM and DVR datasets. As expected, the performance of index bounding decreases with increasing $K$ because of decreasing $N/K$ values. The performance is better in the DVR dataset than the SpM dataset, because $N/K$ values are larger in the DVR dataset. In Table 3, values less than 1 indicate that the index bounding scheme achieves non-overlapping index ranges. As seen in the table, scheme C6 reduces the total separator-index range sizes below $N$ for each CCP instance with $K \leq 64$ in both SpM and DVR datasets. These experimental results show that the proposed DP+ algorithm becomes a linear-time algorithm in practice.

The efficiency of the parametric-search approach depends on two critical issues: number of probe calls and

Table 4: Number of probe calls performed by the parametric search algorithms.

| SPARSE-MATRIX DATASET | | | | | | | | DVR DATASET | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CCP instance | | | | | | | | CCP instance | | | | | | |
| name | K | NC- | NC | NC+ | $\epsilon$BS | $\epsilon$BS+ | EBS | name | K | NC- | NC | NC+ | $\epsilon$BS+&BID | EBS |
| | 16 | 177 | 21 | 7 | 17 | 6 | 7 | | 16 | 202 | 18 | 6 | 16 + 1 | 9 |
| | 32 | 352 | 19 | 7 | 17 | 7 | 7 | | 32 | 407 | 19 | 7 | 19 + 1 | 10 |
| NL | 64 | 720 | 27 | 5 | 16 | 7 | 6 | blunt256 | 64 | 835 | 18 | 10 | 21 + 1 | 12 |
| | 128 | 1450 | 40 | 14 | 17 | 8 | 8 | | 128 | 1686 | 25 | 15 | 22 + 1 | 13 |
| | 256 | 2824 | 51 | 14 | 16 | 8 | 8 | | 256 | 3556 | 203 | 62 | 23 + 1 | 11 |
| | 16 | 190 | 20 | 6 | 19 | 7 | 5 | | 16 | 245 | 20 | 8 | 17 + 1 | 9 |
| | 32 | 393 | 25 | 11 | 19 | 9 | 8 | | 32 | 502 | 24 | 13 | 18 + 1 | 10 |
| cre-d | 64 | 790 | 24 | 10 | 19 | 8 | 6 | blunt512 | 64 | 1003 | 23 | 11 | 18 + 1 | 12 |
| | 128 | 1579 | 27 | 10 | 19 | 9 | 9 | | 128 | 2023 | 26 | 12 | 20 + 1 | 13 |
| | 256 | 3183 | 37 | 13 | 18 | 9 | 9 | | 256 | 4051 | 23 | 12 | 21 + 1 | 13 |
| | 16 | 182 | 19 | 3 | 18 | 6 | 5 | | 16 | 271 | 21 | 10 | 16 + 1 | 13 |
| | 32 | 373 | 22 | 8 | 18 | 8 | 7 | | 32 | 557 | 24 | 12 | 18 + 1 | 12 |
| CQ9 | 64 | 740 | 29 | 12 | 18 | 8 | 8 | blunt1024 | 64 | 1127 | 21 | 11 | 18 + 1 | 12 |
| | 128 | 1492 | 40 | 12 | 17 | 8 | 8 | | 128 | 2276 | 28 | 13 | 19 + 1 | 13 |
| | 256 | 2971 | 50 | 14 | 18 | 9 | 9 | | 256 | 4564 | 27 | 16 | 21 + 1 | 15 |
| | 16 | 185 | 21 | 6 | 17 | 6 | 6 | | 16 | 194 | 22 | 9 | 18 + 1 | 7 |
| | 32 | 364 | 20 | 6 | 17 | 6 | 6 | | 32 | 409 | 19 | 8 | 20 + 1 | 12 |
| ken-11 | 64 | 721 | 48 | 9 | 17 | 7 | 7 | post256 | 64 | 823 | 17 | 11 | 22 + 1 | 12 |
| | 128 | 1402 | 61 | 8 | 16 | 7 | 7 | | 128 | 1653 | 19 | 13 | 22 + 1 | 14 |
| | 256 | 2783 | 96 | 7 | 17 | 8 | 8 | | 256 | 3345 | 191 | 102 | 23 + 1 | 13 |
| | 16 | 210 | 20 | 6 | 20 | 5 | 4 | | 16 | 235 | 24 | 9 | 16 + 1 | 10 |
| | 32 | 432 | 26 | 8 | 19 | 5 | 5 | | 32 | 485 | 23 | 12 | 18 + 1 | 11 |
| mod2 | 64 | 867 | 24 | 6 | 19 | 8 | 8 | post512 | 64 | 975 | 22 | 13 | 20 + 1 | 14 |
| | 128 | 1727 | 38 | 7 | 20 | 7 | 7 | | 128 | 1975 | 25 | 17 | 21 + 1 | 14 |
| | 256 | 3444 | 47 | 9 | 20 | 9 | 9 | | 256 | 3947 | 29 | 20 | 22 + 1 | 15 |
| | 16 | 211 | 22 | 6 | 19 | 5 | 4 | | 16 | 261 | 27 | 10 | 17 + 1 | 13 |
| | 32 | 424 | 26 | 5 | 19 | 6 | 6 | | 32 | 538 | 23 | 13 | 19 + 1 | 14 |
| world | 64 | 865 | 30 | 12 | 19 | 9 | 9 | post1024 | 64 | 1090 | 22 | 12 | 17 + 1 | 14 |
| | 128 | 1730 | 44 | 10 | 19 | 8 | 8 | | 128 | 2201 | 25 | 15 | 21 + 1 | 16 |
| | 256 | 3441 | 44 | 11 | 19 | 10 | 10 | | 256 | 4428 | 28 | 16 | 22 + 1 | 16 |
| AVERAGES OVER K | | | | | | | | | | | | | | |
| | 16 | 193 | 20.5 | 5.7 | 18.5 | 5.8 | 5.2 | | 16 | 235 | 22.0 | 8.7 | 16.7 + 1.0 | 10.2 |
| | 32 | 390 | 23.0 | 7.5 | 18.3 | 6.8 | 6.5 | | 32 | 483 | 22.0 | 10.8 | 18.7 + 1.0 | 11.5 |
| | 64 | 784 | 30.3 | 9.0 | 18.0 | 7.8 | 7.3 | | 64 | 976 | 20.5 | 11.3 | 19.3 + 1.0 | 12.7 |
| | 128 | 1564 | 41.7 | 10.2 | 18.0 | 7.8 | 7.8 | | 128 | 1969 | 24.7 | 14.2 | 20.8 + 1.0 | 13.8 |
| | 256 | 3108 | 54.2 | 11.3 | 18.0 | 8.8 | 8.8 | | 256 | 3982 | 83.5 | 38.0 | 22.0 + 1.0 | 13.8 |

cost of each probe call. The dynamic index bounding schemes proposed for the parametric-search algorithms clearly reduce the cost of an individual probe call. Table 4 illustrates the performance of proposed parametric-search algorithms in reducing the number of probe calls. For Table 4, in order to compare the relative performance of EBS with $\epsilon$BS+ on the DVR dataset, we enforced the $\epsilon$BS+ algorithm to find optimal partitions by running it with $\epsilon = w_{min}$ and then improving the resulting partition with the BID algorithm. Column $\epsilon$BS+&BID refers to this scheme, and values after "+" denote the additional bottleneck values tested by the BID algorithm. As seen in Table 4, exactly one final bottleneck-value test was needed by the BID algorithm to reach an optimal partition in each CCP instance.

In Table 4, comparison of NC- and NC columns shows that dynamic bottleneck-value bounding drastically decreases the number of probe calls in Nicol's algorithm. Comparison of $\epsilon$BS and $\epsilon$BS+ columns in the SpM dataset shows that using $B_{RB}$ instead of $W_{tot}$ for the upper bound on the bottleneck values considerably reduces the number of probes. Comparison of $\epsilon$BS+ and EBS columns reveals two different behaviors on the SpM and DVR datasets. Discretized dynamic bottleneck-value bounding used in EBS produces only minor improvement on the SpM dataset because of the already discrete nature of integer-valued workload arrays. However, the effect of discretized dynamic bottleneck-value bounding becomes significant on real-valued workload arrays of the DVR dataset.

Table 5: Partitioning times for the sparse-matrix dataset as percents of SpMxV times.

| SPARSE-MATRIX DATASET | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | HEURISTICS | | | EXACT ALGORITHMS | | | | | | | | |
| CCP instance | | | | | EXISTING | | | | PROPOSED | | | | |
| name | $K$ | H1 | H2 | RB | DP | MS | $\epsilon$BS | NC | DP+ | MS+ | $\epsilon$BS+ | NC+ | BID |
| | 16 | 0.09 | 0.09 | 0.09 | 93 | 119 | 0.93 | 1.20 | 0.40 | 0.44 | 0.40 | 0.40 | 0.22 |
| | 32 | 0.18 | 0.18 | 0.13 | 177 | 194 | 1.77 | 2.17 | 1.73 | 1.73 | 0.80 | 0.80 | 0.44 |
| NL | 64 | 0.35 | 0.35 | 0.31 | 367 | 307 | 3.15 | 5.85 | 5.81 | 4.12 | 1.86 | 1.77 | 1.82 |
| | 128 | 0.89 | 0.84 | 0.71 | 748 | 485 | 6.30 | 17.12 | 19.87 | 26.92 | 4.26 | 7.32 | 4.21 |
| | 256 | 1.51 | 1.55 | 1.37 | 1461 | 757 | 11.09 | 40.31 | 91.80 | 96.41 | 9.80 | 15.70 | 21.06 |
| | 16 | 0.03 | 0.01 | 0.01 | 33 | 36 | 0.33 | 0.37 | 0.12 | 0.06 | 0.10 | 0.11 | 0.03 |
| | 32 | 0.04 | 0.06 | 0.06 | 71 | 61 | 0.65 | 0.93 | 0.47 | 1.20 | 0.29 | 0.33 | 0.10 |
| cre-d | 64 | 0.12 | 0.12 | 0.08 | 147 | 98 | 1.22 | 1.69 | 1.66 | 1.83 | 0.61 | 0.71 | 0.21 |
| | 128 | 0.28 | 0.29 | 0.14 | 283 | 150 | 2.41 | 3.59 | 5.47 | 10.94 | 1.68 | 1.68 | 0.61 |
| | 256 | 0.53 | 0.54 | 0.25 | 571 | 221 | 4.20 | 10.22 | 27.05 | 26.02 | 3.30 | 4.46 | 1.20 |
| | 16 | 0.04 | 0.04 | 0.04 | 59 | 73 | 0.48 | 0.59 | 0.20 | 0.13 | 0.15 | 0.13 | 0.17 |
| | 32 | 0.09 | 0.09 | 0.07 | 127 | 120 | 0.94 | 1.31 | 0.94 | 0.72 | 0.41 | 0.41 | 0.28 |
| CQ9 | 64 | 0.20 | 0.17 | 0.15 | 248 | 195 | 1.85 | 3.18 | 3.01 | 4.47 | 1.00 | 1.44 | 0.68 |
| | 128 | 0.44 | 0.44 | 0.31 | 485 | 303 | 3.18 | 8.52 | 9.65 | 18.82 | 2.44 | 3.05 | 2.51 |
| | 256 | 0.92 | 0.92 | 0.63 | 982 | 469 | 6.51 | 20.33 | 69.56 | 72.2 | 5.32 | 7.45 | 14.92 |
| | 16 | 0.10 | 0.10 | 0.10 | 238 | 344 | 1.27 | 1.88 | 0.56 | 0.61 | 0.46 | 0.41 | 0.25 |
| | 32 | 0.20 | 0.20 | 0.15 | 501 | 522 | 2.29 | 3.10 | 3.82 | 4.78 | 1.22 | 1.17 | 1.63 |
| ken-11 | 64 | 0.46 | 0.46 | 0.36 | 993 | 778 | 4.48 | 13.08 | 9.41 | 24.78 | 3.10 | 3.46 | 2.29 |
| | 128 | 1.17 | 1.17 | 0.71 | 1876 | 1139 | 9.21 | 39.59 | 50.13 | 50.03 | 6.41 | 6.62 | 17.4 |
| | 256 | 2.14 | 2.14 | 1.42 | 3827 | 1706 | 17.76 | 119.13 | 129.01 | 241.48 | 14.91 | 14.50 | 29.11 |
| | 16 | 0.02 | 0.02 | 0.01 | 92 | 119 | 0.27 | 0.34 | 0.07 | 0.05 | 0.06 | 0.06 | 0.03 |
| | 32 | 0.04 | 0.04 | 0.02 | 188 | 192 | 0.51 | 0.78 | 0.19 | 0.18 | 0.16 | 0.15 | 0.07 |
| mod2 | 64 | 0.11 | 0.10 | 0.06 | 378 | 307 | 1.04 | 1.91 | 0.86 | 2.18 | 0.51 | 0.55 | 0.23 |
| | 128 | 0.24 | 0.23 | 0.11 | 751 | 482 | 2.28 | 5.92 | 2.61 | 3.72 | 1.06 | 1.23 | 0.53 |
| | 256 | 0.48 | 0.48 | 0.23 | 1486 | 756 | 4.26 | 11.14 | 12.11 | 50.04 | 2.91 | 3.43 | 3.66 |
| | 16 | 0.02 | 0.02 | 0.02 | 99 | 122 | 0.29 | 0.33 | 0.08 | 0.05 | 0.07 | 0.08 | 0.03 |
| | 32 | 0.04 | 0.04 | 0.04 | 198 | 196 | 0.59 | 0.88 | 0.26 | 0.21 | 0.18 | 0.19 | 0.08 |
| world | 64 | 0.12 | 0.11 | 0.09 | 385 | 306 | 1.16 | 1.70 | 1.09 | 4.84 | 0.64 | 0.54 | 0.35 |
| | 128 | 0.24 | 0.23 | 0.19 | 769 | 496 | 2.19 | 5.32 | 4.48 | 10.15 | 1.31 | 1.23 | 1.77 |
| | 256 | 0.47 | 0.47 | 0.36 | 1513 | 764 | 4.06 | 11.83 | 11.54 | 69.99 | 3.46 | 3.50 | 2.48 |
| AVERAGES OVER K | | | | | | | | | | | | | |
| | 16 | 0.05 | 0.05 | 0.05 | 102 | 136 | 0.60 | 0.78 | 0.24 | 0.22 | 0.21 | 0.20 | 0.12 |
| | 32 | 0.10 | 0.10 | 0.08 | 210 | 214 | 1.12 | 1.53 | 1.23 | 1.47 | 0.51 | 0.51 | 0.43 |
| | 64 | 0.23 | 0.22 | 0.18 | 420 | 332 | 2.15 | 4.57 | 3.64 | 7.04 | 1.29 | 1.41 | 0.93 |
| | 128 | 0.54 | 0.53 | 0.36 | 819 | 509 | 4.26 | 13.34 | 15.37 | 20.1 | 2.86 | 3.52 | 4.51 |
| | 256 | 1.01 | 1.01 | 0.71 | 1640 | 779 | 7.98 | 35.49 | 56.84 | 92.69 | 6.62 | 8.17 | 12.07 |

Tables 5 and 6 display execution times of the CCP algorithms on the SpM and DVR datasets, respectively. In Table 5, execution times are given as percents of single SpMxV times. Actual execution times of the CCP algorithms can easily be recomputed from the SpMxV execution times given in Table 1. For the DVR dataset, actual execution times (in msecs) are dissected into prefix-sum times and partitioning times. In Tables 5 and 6, the execution times of the existing algorithms and their improved versions are listed in the same order under the respective classification, so that amount of improvement in each algorithm can easily be seen. In both tables, BID is listed separately since it is a new iterative-refinement algorithm. The results of both $\epsilon$BS and $\epsilon$BS+ are listed in Table 5, since they are used as exact algorithms on the SpM dataset with $\epsilon = 1$. Since neither $\epsilon$BS nor $\epsilon$BS+ can be used as an exact algorithm on the DVR dataset, EB—being an exact algorithm for general workload arrays—is listed separately in Table 6.

As seen in Tables 5 and 6, the RB heuristic is faster than both H1 and H2 in both SpM and DVR datasets. As also seen in Table 2, RB finds better partitions than both H1 and H2 in both datasets. These results show that RB is a better heuristic than H1 and H2.

In Tables 5 and 6, relative performance comparison of existing exact CCP algorithms shows that NC is two orders of magnitude faster than both DP and MS in both SpM and DVR datasets, and $\epsilon$BS is considerably

Table 6: Partitioning times (in msecs) for the DVR dataset.

| DVR DATASET | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CCP instance | | prefix | HEURISTICS | | | EXACT ALGORITHMS | | | | | | | |
| | | | | | | EXISTING | | | PROPOSED | | | | |
| name | $K$ | sum | H1 | H2 | RB | DP | MS | NC | DP+ | MS+ | NC+ | EBS | BID |
| blunt256 | 16 | 1.95 | 0.02 | 0.02 | 0.03 | 68 | 49 | 0.36 | 0.21 | 0.53 | 0.10 | 0.14 | 0.03 |
| | 32 | | 0.05 | 0.05 | 0.05 | 141 | 77 | 0.77 | 0.76 | 0.76 | 0.21 | 0.27 | 0.24 |
| | 64 | | 0.11 | 0.12 | 0.09 | 286 | 134 | 1.39 | 3.86 | 8.72 | 0.64 | 0.71 | 0.78 |
| | 128 | | 0.24 | 0.25 | 0.20 | 581 | 206 | 3.66 | 12.37 | 29.95 | 1.78 | 1.84 | 2.33 |
| | 256 | | 0.47 | 0.50 | 0.37 | 1139 | 296 | 52.53 | 42.89 | 0.78 | 13.96 | 3.11 | 56.69 |
| blunt512 | 16 | 13.45 | 0.03 | 0.03 | 0.03 | 356 | 200 | 0.55 | 0.25 | 0.91 | 0.14 | 0.16 | 0.09 |
| | 32 | | 0.07 | 0.07 | 0.07 | 792 | 353 | 1.31 | 1.23 | 2.13 | 0.44 | 0.43 | 0.22 |
| | 64 | | 0.18 | 0.19 | 0.14 | 1688 | 593 | 2.65 | 3.68 | 9.28 | 0.94 | 1.10 | 0.45 |
| | 128 | | 0.39 | 0.40 | 0.28 | 3469 | 979 | 5.84 | 15.01 | 46.72 | 2.29 | 2.63 | 1.65 |
| | 256 | | 0.74 | 0.75 | 0.54 | 7040 | 1637 | 9.70 | 57.50 | 164.49 | 5.59 | 5.95 | 10.72 |
| blunt1024 | 16 | 59.12 | 0.03 | 0.03 | 0.03 | 1455 | 780 | 0.75 | 0.93 | 4.77 | 0.25 | 0.28 | 0.19 |
| | 32 | | 0.12 | 0.11 | 0.08 | 3251 | 1432 | 1.81 | 2.02 | 8.92 | 0.60 | 0.62 | 0.31 |
| | 64 | | 0.27 | 0.27 | 0.19 | 6976 | 2353 | 3.50 | 7.35 | 22.28 | 1.27 | 1.37 | 1.39 |
| | 128 | | 0.50 | 0.51 | 0.37 | 14337 | 3911 | 9.14 | 33.01 | 69.38 | 3.36 | 3.56 | 16.21 |
| | 256 | | 0.97 | 1.00 | 0.68 | 29417 | 6567 | 16.59 | 180.09 | 538.10 | 8.48 | 8.98 | 16.29 |
| post256 | 16 | 2.16 | 0.02 | 0.03 | 0.03 | 79 | 61 | 0.46 | 0.18 | 0.13 | 0.10 | 0.11 | 0.24 |
| | 32 | | 0.05 | 0.05 | 0.05 | 157 | 100 | 0.77 | 1.05 | 1.74 | 0.26 | 0.31 | 0.76 |
| | 64 | | 0.10 | 0.11 | 0.10 | 336 | 167 | 1.37 | 5.26 | 9.72 | 0.61 | 0.70 | 4.67 |
| | 128 | | 0.25 | 0.26 | 0.18 | 672 | 278 | 2.90 | 22.94 | 55.10 | 1.65 | 1.77 | 23.96 |
| | 256 | | 0.47 | 0.48 | 0.37 | 1371 | 338 | 45.16 | 84.78 | 0.77 | 13.04 | 3.62 | 299.32 |
| post512 | 16 | 20.55 | 0.02 | 0.02 | 0.03 | 612 | 454 | 0.63 | 0.20 | 0.44 | 0.14 | 0.16 | 1.21 |
| | 32 | | 0.07 | 0.07 | 0.07 | 1254 | 699 | 1.30 | 2.49 | 1.95 | 0.38 | 0.42 | 3.36 |
| | 64 | | 0.18 | 0.18 | 0.13 | 2559 | 1139 | 2.58 | 16.93 | 38.73 | 0.97 | 1.07 | 8.53 |
| | 128 | | 0.38 | 0.39 | 0.28 | 5221 | 1936 | 5.84 | 68.25 | 156.15 | 2.27 | 2.44 | 31.54 |
| | 256 | | 0.70 | 0.73 | 0.53 | 10683 | 3354 | 12.67 | 256.48 | 726.54 | 5.44 | 5.39 | 140.88 |
| post1024 | 16 | 69.09 | 0.03 | 0.04 | 0.03 | 2446 | 1863 | 0.91 | 2.88 | 5.73 | 0.17 | 0.21 | 2.63 |
| | 32 | | 0.09 | 0.08 | 0.08 | 5056 | 2917 | 1.61 | 13.57 | 17.37 | 0.51 | 0.58 | 12.73 |
| | 64 | | 0.25 | 0.26 | 0.17 | 10340 | 4626 | 3.56 | 35.05 | 33.25 | 1.35 | 1.47 | 32.53 |
| | 128 | | 0.51 | 0.53 | 0.36 | 21102 | 7838 | 7.90 | 156.34 | 546.92 | 2.95 | 3.28 | 76.96 |
| | 256 | | 0.95 | 0.98 | 0.67 | 43652 | 13770 | 16.30 | 764.59 | 1451.87 | 7.55 | 7.02 | 300.70 |
| AVERAGES (normalized w.r.t. RB times) OVER K (prefix-sum time not included) | | | | | | | | | | | | | |
| | 16 | | 0.83 | 0.94 | 1.00 | 27863 | 18919 | 20.33 | 25.83 | 69.50 | 5.00 | 5.89 | 24.39 |
| | 32 | | 1.10 | 1.06 | 1.00 | 23169 | 12156 | 18.47 | 47.37 | 72.82 | 5.83 | 6.46 | 39.02 |
| | 64 | | 1.30 | 1.35 | 1.00 | 22636 | 9291 | 17.88 | 82.81 | 145.19 | 7.00 | 7.81 | 53.81 |
| | 128 | | 1.35 | 1.39 | 1.00 | 22506 | 7553 | 20.46 | 168.36 | 481.19 | 8.60 | 9.31 | 86.81 |
| | 256 | | 1.35 | 1.39 | 1.00 | 24731 | 6880 | 59.10 | 390.25 | 772.99 | 19.55 | 10.51 | 286.77 |
| AVERAGES (normalized w.r.t. RB times) OVER K (prefix-sum time included) | | | | | | | | | | | | | |
| | 16 | | 1.00 | 1.00 | 1.00 | 32 | 23 | 1.08 | 1.04 | 1.09 | 1.01 | 1.02 | 1.03 |
| | 32 | | 1.00 | 1.00 | 1.00 | 66 | 36 | 1.15 | 1.21 | 1.29 | 1.04 | 1.05 | 1.13 |
| | 64 | | 1.00 | 1.00 | 1.00 | 135 | 58 | 1.27 | 1.97 | 2.90 | 1.11 | 1.12 | 1.55 |
| | 128 | | 1.01 | 1.01 | 1.00 | 275 | 94 | 1.62 | 4.75 | 10.53 | 1.28 | 1.30 | 3.25 |
| | 256 | | 1.02 | 1.02 | 1.00 | 528 | 142 | 7.98 | 14.64 | 13.71 | 2.95 | 1.55 | 26.73 |

faster than NC on the SpM dataset. These results show that among the existing algorithms parametric-search approach leads to faster algorithms than both dynamic-programming and iterative-improvement approaches.

Tables 5 and 6 show that our improved algorithms are significantly faster than the respective existing algorithms. In dynamic-programming approach, DP+ is two-to-three orders of magnitude faster than DP such that DP+ competes with the parametric-search algorithms. In the SpM dataset, DP+ is 630 times faster than DP on average in 16-way partitioning, and this ratio decreases to 378, 189, 106, and 56 with increasing $K$ values, 32, 64, 128, and 256, respectively. In the DVR dataset, if initial prefix-sum is not included, DP+ is 1277, 578, 332, 159, and 71 times faster than DP for $K = 16, 32, 64, 128,$ and 256, respectively, on average. This decrease is expected since effectiveness of separator-index bounding decreases with increasing $K$. These experimental findings agree with the variation of the effectiveness of separator-index bounding values displayed in Table 3. In iterative refinement approach, MS+ is also one-to-three orders of magnitude faster than MS, where this drastic improvement simply depends on the scheme used for finding an initial leftist partition.

As seen in Tables 5 and 6, significant improvement ratios are also obtained in the parametric search algorithms. On average, NC+ is 4.2, 3.5, 3.1, 3.7, and 3.7 times faster than NC for $K = 16, 32, 64, 128$, and 256, respectively, in the SpM dataset. In the DVR dataset, if initial prefix-sum time is not included, NC+ is 4.2, 3.2, 2.6, 2.5, and 2.7 times faster than NC for $K = 16, 32, 64, 128$, and 256, respectively. In the SpM dataset, $\epsilon$BS+ is 3.4, 2.5, 1.8, 1.6, and 1.3 times faster than $\epsilon$BS for $K = 16, 32, 64, 128$, and 256, respectively. These improvement ratios on the execution times of the parametric search algorithms are below the improvement ratios on the numbers of probe calls displayed in Table 4. Overhead due to the RB call and initial settings of the separator indices contributes to this difference in both NC+ and $\epsilon$BS+. Furthermore, costs of initial probe calls with very large bottleneck values are very cheap in $\epsilon$BS.

In Table 5, relative performance comparison of the proposed exact CCP algorithms shows that BID is the clear winner for small to moderate $K$ values (i.e., $K \leq 64$) in the SpM dataset. The relative performance of BID degrades with increasing $K$ so that both $\epsilon$BS+ and NC+ begin to run faster than BID for $K \geq 128$ in general, where $\epsilon$BS+ becomes the winner. In the DVR dataset, NC+ and EBS are clear winners as seen in Table 6. NC+ runs slightly faster than EBS for $K \leq 128$, however EBS runs considerably faster than NC+ for $K = 256$. BID can compete with these two algorithms only in 16 and 32 way partitioning of grid *blunt-fin* (for all mesh resolutions). As seen in Table 5, BID takes less than 1 percent of a single SpMxV time for $K \leq 64$ on average. In the DVR dataset (Table 6), the initial prefix-sum time is considerably larger than the actual partitioning time of the EBS algorithm in all CCP instances except 256-way partitioning of *post256*. As seen in Table 6, EBS is only 12 percent slower than the RB heuristic in 64-way partitionings on average. These experimental findings show that the proposed exact CCP algorithms should replace heuristics.

## 7 Conclusion

We proposed runtime efficient chains-on-chains partitioning algorithms for optimal load balancing in one-dimensional decomposition of nonuniform computational domains. Our main idea was to run an effective heuristic, as a pre-processing step, to find a "good" upper bound on the optimal bottleneck value, and then exploit the lower and upper bounds on the optimal bottleneck value to restrict the search space for separator-index values. This separator-index bounding scheme was exploited in a static manner in the dynamic-programming algorithm drastically reducing the number of table entries computed and referenced. A dynamic separator-index bounding scheme was proposed for parametric search algorithms to narrow separator-index ranges after each probe call. We enhanced the approximate bisection to be an exact algorithm by updating lower and upper bounds into realizable values after each probe call. We proposed a new iterative-refinement scheme, which is very fast for small to medium number of processors. We also showed that the proposed algorithms can be used for heterogeneous parallel systems with minor modifications, and proved the NP-Completeness of the chains partitioning problem for heterogeneous systems, where processor permutation is allowed in subchain to processor assignment.

We investigated two distinct application domains for experimental performance evaluation: 1D decomposition of irregularly sparse matrices for parallel matrix-vector multiplication, and decomposition for image-space parallel volume rendering. Experiments on the sparse matrix dataset showed that 64-way decompositions can be achieved in 100 times less than a single matrix vector multiplication time, while producing 4 times better load imbalance values than the most effective heuristic, on average. Experimental results on the volume rendering dataset showed that exact algorithms can produce 3.8 times better 64-way decompositions than the most effective heuristic, while being only 11 percent slower, on average.

## Appendix A
## A Better Load Balancing Model for Iterative Solvers

Load balancing problem in parallel iterative solvers has been mostly stated by considering only the SpMxV computations since they constitute the most time consuming operation. However, linear vector operations (i.e., DAXPY and inner-product computations) involved in iterative solvers may have a considerable effect on the parallel performance with increasing sparsity of the coefficient matrix. In this appendix, we consider incorporating linear vector operations into the load-balancing model as much as possible.

For the sake of discussion, we will investigate this problem for the coarse-grain formulation [2, 3, 37] of the well-known *conjugate-gradient* (CG) algorithm. In this CG algorithm, each iteration involves, in order of computational dependency, one SpMxV, two inner-product, and three DAXPY computations. DAXPY computations do not involve communication, whereas inner-product computations necessitate a post global *reduction* operation [23] on the results of the two local inner-product results. Note that load balancing problem can only be defined precisely between two successive synchronization points. In rowwise striping, the pre-communication operation needed for SpMxV and the global reduction operation constitute the pre and post synchronization points, respectively, for the aggregate of one local SpMxV and two local inner-product computations. In columnwise striping, the global reduction operation in the current iteration and post-communication operation needed for SpMxV in the next iteration constitute the pre and post synchronization points, respectively, for the aggregate of three local DAXPY and one local SpMxV computations. So, columnwise striping may be favored for a wider coverage of load balancing. Each vector entry incurs a multiply-and-add operation during each linear vector operation. Hence, DAXPY computations can easily be incorporated into the load-balancing model for the CS scheme, by adding a constant cost of 3 to the computational weight $w_i$ of the atomic task $t_i$ representing column $i$ of matrix $A$. Initial prefix-sum operation needed in the CCP algorithms can still be avoided by computing a subchain weight $W_{i,j}$ as $W_{i,j} = \text{COL}[j+1] - \text{COL}[i] + 3(j-i+1)$ in constant time. Note that two local inner-product computations still remain uncovered in this balancing model.

## Appendix B
## Partitioning Chains for Heterogeneous Systems

This paper focused on chain partitioning algorithms for homogeneous parallel systems, i.e., the execution time of each task is equal on all processors. However, heterogeneous systems have growing importance because of growing popularity of build-it-yourself parallel computers, epitomized by Beowulf-class machines [38].

```
HPROBE($B$)
  $s_0 \leftarrow 0$;   $k \leftarrow 1$;
  $Bsum \leftarrow B \times e_1$;
  while $p \leq K$ and $Bsum < W_{tot}$ do
    $s_k \leftarrow$ BINSRCH $(\mathcal{W}, s_{k-1}, N, Bsum)$;
    $Bsum \leftarrow \mathcal{W}[s_k] + B \times e_k$;
    $k \leftarrow k + 1$;
  if $Bsum \geq W_{tot}$ then
    return TRUE ;
  else
    return FALSE;
```

Figure 11: Probe function modified for heterogeneous processors.

Parallel systems built in this way are very likely to exhibit processor heterogeneity. Two distinct problems arise in partitioning chains for heterogeneous systems, referred to here as chains-on-chains partitioning problem for heterogeneous systems (CCP-HET) and chain partitioning problem for heterogeneous systems (CP-HET). In CCP-HET problem, a chain of tasks is to be mapped onto a chain of processors, i.e., $k$th subchain in a partition is assigned to the $k$th processor. In CP-HET problem, a chain of tasks is to be mapped onto a *set*—as opposed to a chain—of processors, i.e., the processors can be permuted for subchain assignment. Following two sections discuss these two problems.

## B.1 CCP Problem on Heterogeneous Systems

In the CCP-HET problem, a chain $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$ of $N$ tasks, with weights $\mathcal{W} = \langle w_1, w_2, \ldots, w_N \rangle$ is to be mapped onto a chain $\mathcal{P} = \langle P_1, P_2, \ldots, P_K \rangle$ of $K$ processors with associated execution speeds $\mathcal{E} = \langle e_1, e_2, \ldots, e_K \rangle$. The execution time of task $t_i$ on processor $P_k$ is equal to $w_i/e_k$. Cost $C(\Pi)$ of a partition $\Pi$ is determined by the maximum execution time among all processors. The objective is to find a partition $\Pi = \{s_0 = 0, s_1, \ldots, s_K = N\}$, where $s_k \leq s_{k+1}$ for $k = 0, 1, \ldots K-1$, with minimum cost

$$C(\Pi) = \max_{1 \leq k \leq K} \frac{W_{s_{k-1}+1, s_k}}{e_k}$$

All algorithms presented in Sections 3 and 4 can be enhanced to optimally solve HET-CCP, without altering the complexities. For HET-CCP algorithms, the weight of a task should be replaced with the execution time of a task on a processor. We will briefly describe guidelines for some of the algorithms due to space considerations.

We can bound the value of an optimal solution for HET-CCP, using the same techniques we did for CCP. The lower bound $LB$ is achieved when all processors are equally loaded, i.e., $LB = W_{tot} / \sum_{k=1}^{K} e_k$. The upper bound $UB$ can be found in practice with a fast and effective heuristic. We can give $UB = LB + w_{max} / \min_{k=1}^{K} e_k$ as a theoretically robust bound for many greedy heuristics. As a constructive proof for this result: assume each processor is minimally loaded to surpass $LB$. The worst case occurs when we have to include the maximally weighted task to surpass $LB$, and this happens for the slowest processor. Notice that the load of the last processor will always be less than or equal to $LB$.

A probe function HPROBE($B$) for heterogeneous systems is presented in Fig. 11. In this algorithm, the execution speed of each processor is enrolled in before binary search, when we compute the load of a processor. We scale the total weight of tasks to be assigned to a processor, proportional to its execution speed. This probe function can be used to bound separator indices according to Lemma 2 and Corollary 6.

The recursion given in (1) for the dynamic programming solution can be restated as:

$$B_i^k = \min_{k-1 \leq j < i} \{\max\{B_j^{k-1}, \frac{W_{j+1,i}}{e_k}\}\}$$

This equation defines an $O(N^2K)$ algorithm, but the observations given in Section 3.2 for reducing the complexity to $O(NK)$ are still valid. All that needs to be done is to replace subchain weight computations $\mathcal{W}[i] - \mathcal{W}[j]$ in the DP and DP+ algorithms presented in Figs. 1 and 6, with $(\mathcal{W}[i] - \mathcal{W}[j])/e_k$.

To enhance the bidding algorithm for heterogeneous systems, we have to be able to compute the next smallest candidate bottleneck value, and move the separators according to this new value. The bid of a processor $P_k$ can be computed as the total chain weight with one more task added, divided by the execution speed $e_k$. Let $\Pi = \{s_0, s_1, \ldots, s_K\}$ be the current solution. The bid of processor $P_k$ is $W_{s_{k-1}+1, s_k+1} \, / \, e_k$ for $k = 1, 2, \ldots, K-1$. The bid of the last processor can be computed as the remaining task weight divided by the execution speed of the last processor, i.e., $W_{s_{K-1}+1, N} \, / \, e_K$. Once we decide on the next candidate bottleneck value, we have to adjust the separator indices by adding/removing tasks to/from processors. While adding (removing) a task to (from) a processor we add (subtract) the weight of the task divided by the execution speed of the respective processor.

The lower and upper bounds together with the HPROBE($B$) function is sufficient for the $\epsilon$-approximate bisection algorithm. For the exact bisection algorithm, we need to move upper and lower bounds to realizable bottleneck values. As in CCP, our probe function constructs a solution, hence we already have a realizable upper bound when the probe succeeds. When the probe fails, we can use the bid value that is described in the previous paragraph. Therefore we can restrict our search space to realizable bottleneck values. Each probe call helps us to eliminate at least one candidate bottleneck value, the bisection algorithm terminates in finite number of steps to find the optimal solution value.

The proposed techniques can be enhanced further to solve the CCP problem, when execution time of a task is an arbitrary function of the processor that the task is assigned to (provided that the execution time is positive). The presented algorithms can still be adapted for this problem without altering their complexities. In this case, we no longer have a definition of weight of a task $w_i$, but have definition of weight of a task $t_i$ for a processor $P_k$, $w_i^k$, which should be used for computing the load of a processor.

## B.2 Chains Partitioning for Heterogeneous Systems

The solution to this problem is not only the separators, but also processor assignments for subchains. Thus, we define a mapping $\mathcal{M}$ as a partition $\Pi = \langle s_0 = 0, s_1, \ldots, s_K = N \rangle$ of the given task chain $\mathcal{T} = \langle t_1, t_2, \ldots t_N \rangle$ with $s_k \leq s_{k+1}$ for $k = 0, 1, \ldots, K-1$, and a permutation $\langle \pi_1, \pi_2, \ldots, \pi_K \rangle$ of the given set of $K$ processors $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$. According to this mapping, $k$th task subchain $\langle t_{s_{k-1}+1}, \ldots, t_{s_k} \rangle$ is executed on processor $P_{\pi_k}$. Cost $C(\mathcal{M})$ of a mapping $\mathcal{M}$ is the largest subchain computation time, which is determined by the subchain weight and the execution speed of the assigned processor, i.e.,

$$C(\mathcal{M}) = \max_{1 \leq k \leq K} \frac{W_{s_{k-1}+1, s_k}}{e_{\pi_k}}$$

Below we provide a formal definition of the decision problem, followed by NP-Completeness proof.

*Given a chain of tasks $\mathcal{T} = \langle t_1, t_2, \ldots t_N \rangle$, a weight $w_i \in Z^+$ for each $t_i \in \mathcal{T}$, a set of processors $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$ such that $K < N$ and an execution rate $e_k \in Z^+$ for each $P_k \in \mathcal{P}$. Decide if there exists a mapping $\mathcal{M}$ of chain $\mathcal{T}$ onto set of processors $\mathcal{P}$ such that $C(\mathcal{M})$ is less than or equal to a specified value.*

LEMMA B1. *The chain-partition problem is NP-Hard.*

PROOF. We will use reduction from 3-Partition (3P) problem. A pseudo-polynomial transformation suffices, because 3P problem is NP-Complete in the strong sense (i.e., there is no pseudo-polynomial time algorithm for the problem unless P=NP). The 3P problem is stated in [10] as follows.

*Given a finite set $\mathcal{A}$ of $3m$ elements, a bound $B \in Z^+$, and a cost $c_i \in Z^+$ for each $a_i \in \mathcal{A}$, such that each $c_i$ satisfies $B/4 < c_i < B/2$ and such that $\sum_{a_i \in \mathcal{A}} c_i = mB$. Decide if $\mathcal{A}$ can be partitioned into $m$ disjoint sets $S_1, S_2, \ldots, S_m$ such that $\sum_{a_i \in S_k} c_i = B$* for $k = 1, 2, \ldots, m$ .

For a given instance of the 3P problem, corresponding chain partitioning problem is constructed as follows.

- The number of tasks $N$ is $m(B+1)-1$. The weight of every $(B+1)$st task is $B$, (i.e., $w_i = B$ for $i \bmod (B+1) = 0$), and the weights of all other tasks are 1.
- The number of processors $K$ is $4m-1$. The first $m-1$ processors have execution speeds of $B$, (i.e., $e_k = B$ for $k = 1, 2, \ldots, m-1$), and the remaining processors have execution speeds equal to costs of items in the 3P problem (i.e., $e_k = c_{k-m+1}$ for $k = m, \ldots, 4m-1$).

We claim that there is a solution to the 3P problem if there is a mapping $\mathcal{M}$ with cost $C(\mathcal{M}) = 1$ for the chain partitioning problem. The following observations constitute the basis for our proof.

- The processors with execution speeds of $B$ must be mapped to tasks with weight $B$ to have a solution with cost $C(\mathcal{M}) = 1$, because the execution speeds of all other processors are less than or equal to $B/2$. These processors (tasks) serve as divider processors (tasks).
- The total weight of the chain is $3m + (m-1)B = (B+3)m - B$. The sum of the execution speeds of all processors is also $(m-1)B + 3m = (B+3)m - B$. This forces each processor to be assigned a load with value equal to its execution speed, to achieve a mapping with cost $C(\mathcal{M}) = 1$. This observation can be generalized: If a subchain of tasks with total weight $W$ is mapped to a subset of processors with total execution speed $E$, then $E = W$, to have a solution with $C(\mathcal{M}) = 1$. Notice that the reverse direction of the proposal also holds thanks to the unit-weight tasks between the dividers.

As noted above, divider processors should be assigned to divider tasks. Between two successive divider tasks there is a subchain of $B$ unit-weight tasks with total weight $B$, which must be assigned to a subset of processors with total execution speed $B$. Since there are $m$ such subchains, the same grouping is also valid for grouping $c_i$ values in the 3P problem. $\square$

THEOREM B2. *The chain partition problem is NP-Complete.*

PROOF. The cost of a given mapping can be computed in polynomial time, thus the problem is in NP. With the result of Lemma B1, we can conclude that the chain partition problem is NP-Complete. $\square$

## References

[1] S. Anily and A. Federgruen, Structured partitioning problems, *Operations Research* **13**, (1991), 130–149.

[2] C. Aykanat and F. Ozguner, Large Grain Parallel Conjugate Gradient Algorithms on a Hypercube Multiprocessor, *in* "Proc. 1987 Int. Conf. on Parallel Processing," pp. 641–645, 1987.

[3] C. Aykanat, F. Ozguner, F. Ercal, and P. Sadayappan, Iterative algorithms for solution of large sparse systems of linear equations on hypercubes, *IEEE Trans. on Computers* **37**, 12 (Dec. 1988), 1554–1567.

[4] S. H. Bokhari, Partitioning problems in parallel, pipelined, and distributed computing, *IEEE Trans. on Computers* **37**, 1 (1988), 48–57.

[5] Ü. V. Çatalyurek and C. Aykanat, Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication, *IEEE Trans. on Parallel and Distributed Systems* **10**, 7 (1999), 673–693.

[6] H. Choi and B. Narahari, Algorithms for mapping and partitioning chain structured parallel computations, *in*, "Proc. 1991 Int. Conf. on Parallel Processing," pp. I-625–I-628, 1991.

[7] G. N. Frederickson, "Optimal algorithms for partitioning trees and locating $p$-centers in trees," Purdue Univ. Tech. Rep. CSD-TR-1029, Purdue University, 1990.

[8] G. N. Frederickson, Optimal algorithms for partitioning trees. *in*, "Proc. Second ACM-SIAM Symp. Discrete Algorithms," pp. 168–177, 1991.

[9] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified NP-complete graph problems," *Theoretical Computer Science*, vol. 1, pp. 237–267, 1976.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, San Fransisco, U.S.A.: W. H. Freeman, 1979.

[11] D. M. Gay, Electronic mail distribution of linear programming test problems, *Mathematical Programming Society COAL Newsletter*, (1985).

[12] Y. Han, B. Narahari, and H.-A. Choi, Mapping a chain task to chained processors, *Infor. Proc. Let.* **44**, (1992), 141–148.

[13] P. Hansen and K. W. Lih, Improved algorithms for partitioning problems in parallel, pipelined and distributed computing, *IEEE Trans. Computers* **41**, 6 (June 1992), 769–771.

[14] B. Hendrickson and T.G. Kolda, Partitioning Rectangular and Structurally Nonsymmetric Sparse Matrices for Parallel Processing, *SIAM J. Scientific Computing*, to appear.

[15] IOWA Optimization Center, Linear programming problems, ftp://col.biz.uiowa.edu:pub/testprob/lp/gondzio.

[16] M. A. Iqbal, "Approximate algorithms for partitioning and assignment problems," ICASE Rep. No. 86-40, NASA Contractor Report 178130, June 1986.

[17] M. A. Iqbal, J. H. Saltz, and S. H. Bokhari, A comparative analysis of static and dynamic load balancing strategies, *in* "Proc. 1986 Int. Conf. on Parallel Processing," pp. 1040–1047, 1986.

[18] M. A. Iqbal, Efficient Algorithms for Partitioning Problems. *in* "Proc. 1990 Int. Conf. on Parallel Processing (ICPP'90)," pp. III-123–III-127, 1990.

[19] M. A. Iqbal and S. H. Bokhari, "Efficient algorithms for a class of partitioning problems," ICASE Report No. 90-49, July 1990.

[20] M. A. Iqbal, Approximate algorithms for partitioning and assignment problems. *Int. J. Parallel Programming* **20**, 5 (1991).

[21] M. A. Iqbal and S. H. Bokhari, Efficient algorithms for a class of partitioning problems, *IEEE Trans. Parallel and Distributed Systems* **6**, 2 (1995), 170–175.

[22] D. M. Kincaid, D. M. Nicol, D. Shier, and D. Richards, A multistage linear array assignment problem, *Operations Research* **38**, 6 (1990), 993–1005.

[23] V. Kumar, A. Grama, A. Gupta, and G. Karypis, "Introduction to Parallel Computing," Benjamin/Cummings, Redwood City, California, 1994.

[24] H. Kutluca, T. M. Kurç, and C. Aykanat, Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids, *J. Supercomputing*, **15**(1), (2000), 51–93.

[25] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. Chichester, U.K.: Wiley, 1990.

[26] F. Manne and T. Sørevik, Optimal partitioning of sequences, *J. Algorithms* **19**, (1995), 235–249.

[27] S. Miguet and J. M. Pierson, Heuristics for 1D rectilinear partitioning as a low cost and high quality answer to dynamic load balancing, *Lecture Notes in Computer Science* **1225**, (1997), 550–564.

[28] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, A sorting classification of parallel rendering, *IEEE Computer Graphics and Applications* **14**, 4 (1994), 23–32.

[29] C. Mueller, The sort-first rendering architecture for high-performance graphics, *in* "Proc. 1995 Symposium on Interactive 3D Graphics," pp. 75–84, 1995.

[30] D. M. Nicol and D. R. O'Hallaron, Improved algorithms for mapping pipelined and parallel computations. ICASE Report 88-2, NASA Contractor Report 181655, April 1988.

[31] D. M. Nicol and D. R. O'Hallaron, Improved algorithms for mapping pipelined and parallel computation, *IEEE Trans. Computers* **40**, 3 (1991), 295–306.

[32] D. M. Nicol, Rectilinear partitioning of irregular data parallel computations. ICASE Report 91-55, NASA Contractor Report 187601, July 1991.

[33] D. M. Nicol, Rectilinear partitioning of irregular data parallel computations, *J. Parallel and Disributed Computing* **23**, (1994), 119–134.

[34] B. Olstad and F. Manne, Efficient partitioning of sequences, *IEEE Trans. Computers* **44**, 11 (1995), 1322–1326.

[35] J. R. Pilkington and S. B. Baden, Dynamic partitioning of non-uniform structured workloads with spacefilling curves, *IEEE Trans. Parallel and Distributed Systems* **7**, 3 (1996), 288–299.

[36] L. F. Romero and E. L. Zapata, Data distributions for sparse matrix vector multiplication, *Parallel Computing* **21**, (1995), 583–605.

[37] Y. Saad, Practical use of polynomial preconditionings for the conjuagte gradient method, *SIAM J. Scientific and Statistical Computing* **6**, 5 (1985), 865–881.

[38] T. Sterling, et al., Beowulf: A Parallel Workstation for Scientific Computation. *in* "Proc. 1995 Int. Conf. on Parallel Processing (ICPP'95)," pp. I-11–I-14, 1995.

[39] M. U. Ujaldon, E. L. Zapata, S. D. Sharma, and J. Saltz, Parallelization techniques for sparse matrix applications, *J. Parallel and Distributed Computing* **38**, (1996), 256–266.

[40] M. U. Ujaldon, E. L. Zapata, S. D. Sharma, and J. Saltz, Experimental evaluation of efficient sparse matrix distributions, *in* "Proc. ACM Int. Conf. of Supercomputing," pp. 78–86, 1996.

[41] M. U. Ujaldon, E. L. Zapata, B. M. Chapman, and H. P. Zima, Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation, *IEEE Trans. Parallel and Distributed Systems* **8**, 10 (Oct. 1997), 1068–1083.